Zur Berechnung von Klassengruppen

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

von

Dipl.-Inf. Stefan Neis

aus Wadern (Saarland)

Referenten:	Prof. Dr. J. Buchmann
	Prof. Dr. A. Pethő
Tag der Einreichung:	4.4.2002
Tag der mündlichen Prüfung:	28.5.2002

Darmstadt 2002
D 17

An dieser Stelle möchte ich allen danken, die mich während der Entstehung dieser Dissertation unterstützten.

Mein Dank gebührt zunächst den Mitgliedern der Prüfungskommission, insbesondere Herrn Prof. Dr. Buchmann, der mein Interesse am Problem der Klassengruppenberechnung geweckt hat, und in zahlreichen Diskussionen immer wieder Anregungen für die Lösung von Teilproblemen geliefert hat, sowie Herrn Prof. Dr. Pethő für die Übernahme des Koreferats.

Besonderer Dank gilt auch Dr.-Ing. Patrick Theobald und Dr. Damian Weber, die im Rahmen ihrer Dissertationen Algorithmen untersucht und implementiert haben, die sich als für mein Gesamtverfahren wesentliche Bausteine erwiesen. Daneben gilt natürlich auch der gesamten LiDIA–Gruppe mein Dank, die mir ihrer Software das Fundament für meine Implementierung geliefert hat, ferner möchte ich Dr.-Ing. Patrick Theobald und Andreas Meyer für das Korrekturlesen meiner Arbeit danken.

Dank gilt auch meinen Kollegen Dr.-Ing. Susanne Wetzel, Klaus Kiefer und nochmals Dr.-Ing. Patrick Theobald für die gute Arbeitsatmosphäre.

Ebenso gilt mein Dank meinen Kollegen der Firma Kobil Systems GmbH und insbesondere deren Inhaber Herrn Ismet Koyun für Geduld und Verständnis, wenn dieser Dissertation wegen die Firma gelegentlich etwas zu kurz kam.

Schließlich gilt mein Dank insbesondere meinen Eltern, die mir das Studium ermöglichten und mich nach besten Kräften unterstützten.

Ferner möchte ich noch die Free Software Foundation erwähnen, deren Software wesentlich zur Erleichterung meiner Arbeit beigetragen hat, Dank gilt dabei insbesondere Eberhard Mattes, Kai Uwe Rommel und Dr. Holger Veit, die mit ihren OS/2–Portierungen die für mich nützlichsten Versionen dieser und anderer Unix–Software zur Verfügung gestellt haben.

Inhaltsverzeichnis

Einleitung

Zu den wichtigen Problemen der algorithmischen algebraischen Zahlentheorie gehören die Berechnung der Klassengruppe und des Regulators bzw. eines Systems von Fundamentaleinheiten eines algebraischen Zahlkörpers. Dabei ist man insbesondere an Verfahren interessiert, die in der Praxis ein gutes Laufzeitverhalten aufweisen.

An effizienten Algorithmen zur Klassengruppenberechnung wird schon seit etlichen Jahren geforscht. Bisher gibt es neben exponentiellen Algorithmen einen subexponentiellen Algorithmus, den Hafner und McCurley für Zahlkörper vom Grad 2 in [HM89] vorgeschlagen haben, und den Buchmann in [Buc89] für höhere Grade verallgemeinert hat. Bei der Implementierung dieser Algorithmen stößt man immer wieder auf die gleichen Teilprobleme. Zunächst benötigt man eine Arithmetik für Ideale, sodann muss man Zerlegungen von Hauptidealen in Primideale (sogenannte Relationen) finden und zuletzt muss man mit Methoden der Linearen Algebra aus den gefundenen Relationen die gewünschten Informationen konstruieren. In dieser Arbeit wollen wir uns mit den ersten beiden Teilproblemen auseinandersetzen, den letzten Teilschritt behandelt Theobald in [The00].

Dabei zeigen wir, dass man für kleine Körpergrade (wir betrachten die Grade 3 bis 6) auch noch für große Diskriminanten (bis zu etwa 50 Dezimalstellen) die Klassengruppe im Sinne eines unter Annahme der verallgemeinerten Riemannschen Vermutung bewiesenen Ergebnisses berechnen kann.

Dazu betrachten wir zunächst die Idealarithmetik. Hierfür gibt es bereits praktische Lösungen, die etwa von Cohen in [Coh95], Kapitel 4.7 beschrieben und in den Systemen KaSh und PARI implementiert werden. Diese leiden jedoch darunter, dass es sich um historisch gewachsene und immer wieder erweiterte und optimierte Ad-hoc-Methoden handelt, mit denen der jeweilige Autor sein spezifisches Problem lösen wollte. Dadurch ergibt sich, dass die einzelnen Ope-

rationen völlig unabhängig voneinander beschrieben und implementiert werden, so dass die gemeinsame Grundidee kaum noch erkennbar ist. Für eine objekt–orientierte Implementierung, wie wir sie für LiDIA anstreben, sind sie daher denkbar schlecht geeignet.

Daher erarbeiten und implementieren wir ein sauberes objekt–orientiertes Design, bei dem wir eine systematische Reduktion der Operationen der Idealarithmetik auf Probleme der Linearen Algebra über Ringen $\mathbb{Z}/m\mathbb{Z}$ zeigen, genauer auf die Berechnung von Kernen und Bildern von Homomorphismen über solchen Ringen. Damit entwickeln wir eine besonders übersichtliche und verständliche Methode der Idealarithmetik, die zudem – wie wir an einigen Beispielen zeigen werden – vergleichbare Effizienz bietet. Dabei müssen wir natürlich insbesondere auch die Probleme der Linearen Algebra über $\mathbb{Z}/m\mathbb{Z}$ lösen. Deshalb zeigen wir erstmals im Detail, wie dies für den Fall von zusammengesetzten Zahlen m möglich ist, wobei wir auf einer Idee von Howell aus [How86] aufbauen.

Danach wenden wir uns der Relationengenerierung zu. Die Grundidee der Verfahren von Hafner und McCurley sowie Buchmann ist dabei mit Dixons Random–Square–Methode beim Faktorisieren (siehe [Bre88], Kapitel 8.1) vergleichbar, d.h. es wird versucht, viele zufällige Zahlen durch Probedivision zu faktorisieren. Bei zufälligen Zahlen schlägt dies zum einen oft fehl, die Zahl lässt sich über einer fixierten Menge von Primzahlen nicht zerlegen, zum anderen ist solch eine vollständige Probedivision für eine einzelne Zahl vergleichsweise zeitaufwendig. Im Unterschied zu Dixons Methode, bei der es um eine Zerlegung über $\mathbb{Z}$ geht, benötigen wir eine Zerlegung in ein Produkt von Primidealen über einem fixierten algebraischen Zahlkörper, wodurch die Zerlegung nochmals etwas aufwendiger wird, so dass das Interesse an einer Optimierung noch stärker ist.

Für das Faktorisieren werden aktuell stets Siebverfahren verwendet. Durch diese wurde eine drastische Verringerung der Laufzeit möglich. Man ist daher daran interessiert, die Parallele zwischen dem Faktorisierungsproblem und der Klassengruppenberechnung zu nutzen und diese Verfahren zu übertragen. Eine Übertragung des quadratischen Siebs auf den Fall quadratischer Zahlkörper hat dabei Jacobson in [JJ98] vorgenommen. Diese Übertragung hat sich hervorragend bewährt, ist jedoch für den Fall von Zahlkörpern vom Grad > 2 nicht anwendbar. Daher wollen wir nun das Number–Field–Sieve, den asympto-

tisch besten bekannten Faktorisierungsalgorithmus, dessen Grundidee z.B. in [BLP93] beschrieben wird, auf den allgemeinen Fall übertragen. Dabei kommt uns entgegen, dass das Number–Field–Sieve ohnehin schon Zerlegungen über algebraischen Zahlkörpern berechnet.

Der wesentliche Vorteil dieser Siebverfahren liegt darin, dass man im Gegensatz zur Probedivision gewissermaßen an vielen Zahlen zugleich arbeitet und nach einer gewissen Zeit einfach alle Zahlen ablesen kann, die sich (fast) zerlegen lassen. Für diese vergleichsweise wenigen Zahlen kann man eine Probedivision durchführen, von der man dann a priori weiß, dass sie (mit hoher Wahrscheinlichkeit) erfolgreich sein wird.

Wir zeigen in der vorliegenden Arbeit, wie man die Idee des Number–Field–Sieves auch auf die Klassengruppenberechnung übertragen kann. Wir demonstrieren, dass man mit dieser Methode – wie im Falle der Faktorisierung – sehr viele algebraische Zahlen in vergleichsweise kurzer Zeit zerlegen kann. Anhand praktischer Beispiele illustrieren wir, dass man mit unserem Verfahren Klassengruppen (und Regulatoren) algebraischer Zahlkörper mit bis zu 50-stelliger Diskriminante effizient berechnen kann. Dabei liefert unser Verfahren ein Ergebnis, das unter der Annahme der verallgemeinerten Riemannschen Vermutung korrekt ist.

Der Aufbau dieser Arbeit ist nun der folgende. In Kapitel 1 tragen wir die verwendeten Begriffe der algebraischen Zahlentheorie zusammen und stellen unsere Notation vor. In Kapitel 2 beschäftigen wir uns mit Zahlkörpern und Ordnungen, vor allem mit ihrer Implementierung und einigen zentralen Algorithmen, insbesondere mit der Berechnung des Ganzheitsringes eines Zahlkörpers.

Kapitel 3 ist der Idealarithmetik sowie den Operationen mit $\mathbb{Z}$–Moduln gewidmet. Hier steht unsere systematische Reduktion der Idealarithmetik auf Lineare Algebra über $\mathbb{Z}/m\mathbb{Z}$ im Mittelpunkt des Interesses. Danach werden wir insbesondere zeigen, wie man die Probleme der Linearen Algebra über $\mathbb{Z}/m\mathbb{Z}$ lösen kann. Dazu untersuchen wir allgemeiner, wie sich diese Probleme über Hauptidealringen R algorithmisch lösen lassen. Dabei beschreiben wir insbesondere unsere Implementierung in LiDIA.

In Kapitel 4 gehen wir auf den Spezialfall der Primideale ein. Hier steht im Hinblick auf die Klassengruppenberechnung vor allem die Platzeffizienz der Darstellung im Vordergrund. Schließlich skizzieren wir die benutzten Verfahren zur Zerlegung von Primzahlen in Primideale eines algebraischen Zahlkörpers

und beschreiben, wie wir unter Benutzung dieser Verfahren Ideale in ein Potenzprodukt von Primidealen zerlegen.

In Kapitel 5 schließlich wenden wir uns der Klassengruppenberechnung zu. Ausgehend von einer kurzen Beschreibung bekannter Verfahren zeigen wir zunächst, welche Datenstrukturen wir zur Implementierung verwenden. Aufbauend auf den „klassischen“ Methoden zeigen wir, wie man das Number–Field–Sieve für die Berechnung von Klassengruppen benutzen kann. Dazu zeigen wir zunächst, wie man die Ideen des Number–Field–Sieves nutzen kann, um glatte Stellen von Polynomen zu finden und wie man diese Methode verwenden kann, um Hauptideale aus einem algebraischen Zahlkörper zu zerlegen. Dann zeigen wir, wie man die bei diesen Zerlegungen auftretenden Probleme lösen kann. Insbesondere muss man für jedes Primideal testen können, ob es einen Beitrag zur Klassengruppe liefert, während beim Faktorisieren völlig uninteressant ist, ob ein Primideal etwas dazu beiträgt, die Zahl zu faktorisieren oder nicht. Ein schwieriges Teilproblem ist dabei die Berechnung der Hermite–Normalform von sehr großen, allerdings dünn besetzten Matrizen. Wie man hierbei vorgeht, wird in einer anderen Arbeit beschrieben, ebenso wie die Auswertung der Relationen, um ein System von Fundamentaleinheiten zu bestimmen.

Dabei zeigen wir insbesondere auch Laufzeiten und Ergebnisse, die wir mit unserem Verfahren erhalten haben und zeigen anhand dieser die Effizienz von Siebverfahren.

Kapitel 1

Begriffe und Notationen

Zunächst stellen wir einige Grundlagen der algebraischen Zahlentheorie vor, die wir später benötigen werden. Eine detailliertere Einführung sowie Beweise zu den zitierten Sätzen können z.B. in [Coh95] oder [Neu92] gefunden werden.

1.1 Algebraische Zahlkörper und algebraische Zahlen

1.1.1 Definitionen

1.1.1. Definition

Ein Polynom $A(x) = \sum_{i=0}^{n} a_i x^i \in \mathbb{Q}[x]$ *heißt* irreduzibel *über* $\mathbb{Q}$*, wenn es keine Polynome* $B(x), C(x) \in \mathbb{Q}[x]$ *von positivem Grad mit* $A(x) = B(x)C(x)$ *gibt.*

Für solch ein irreduzibles Polynom $A(x)$ *ist* $\mathbb{Q}[x]/(A(x)\mathbb{Q}[x])$ *ein Körper, der isomorph ist zu* $\mathbb{Q}(\rho)$*, wobei* ρ *eine sogenannte* formale Nullstelle *des Polynoms* A *ist, d.h. es gilt* $\sum_{i=0}^{n} a_i \rho^i = 0$*. Ein solcher Körper* $\mathbb{Q}(\rho)$ *heißt* algebraischer Zahlkörper*, die Zahlen aus* $\mathbb{Q}(\rho)$ *heißen* algebraisch *und falls* $a_n \neq 0$*, so heißt* a_n *der* Leitkoeffizient *von* A*, in Zeichen* $\mathrm{lc}(A)$ *und* n *heißt* Körpergrad*. Der Körpergrad eines Körpers* $\mathbb{Q}(\rho)$*, in Zeichen* $[\mathbb{Q}(\rho) : \mathbb{Q}]$ *gibt zugleich an, welche Dimension* $\mathbb{Q}(\rho)$ *als* $\mathbb{Q}$*–Vektorraum hat.*

1.1.2. Definition

Sei α *eine algebraische Zahl. Das normierte Polynom* $m_\alpha \in \mathbb{Q}[x]$ *von kleinstmöglichem Grad, das* α *als Nullstelle hat, heißt* Minimalpolynom zu α*. Eine Zahl, deren Minimalpolynom Koeffizienten in* $\mathbb{Z}$ *hat, heißt* (algebraisch) ganze Zahl.

Im Falle eines algebraischen Zahlkörpers $\mathbb{Q}(\rho)$, wobei ρ die formale Nullstelle eines irreduzibles Polynom $A(x)$ ist, ist also $\frac{A}{\mathrm{lc}(A)}$ das *Minimalpolynom zu* ρ.

Ein algebraischer Zahlkörper kann in $\mathbb{C}$ eingebettet werden, indem ρ auf eine komplexe Nullstelle ρ_i des Polynoms $A(x)$ abgebildet wird. Diese Abbildung ergibt einen Homomorphismus, wenn man ein Polynom $f(\rho)$ auf $f(\rho_i)$ abbildet. Für einen Zahlkörper vom Grad n gibt es genau n verschiedene Einbettungen, da die Nullstellen eines über $\mathbb{Q}$ irreduziblen Polynoms stets paarweise verschieden sind. Genauer sagt man:

1.1.3. Definition
Sei ρ eine algebraische Zahl und $A(x) = \sum_{i=0}^{n} a_i x^i$ das zugehörige Minimalpolynom. Dann heißen die n Nullstellen $\rho_1, \ldots \rho_n$ des Polynoms zueinander konjugiert*. Die Abbildungen, die ρ auf eine Nullstelle ρ_i des Polynoms abbilden und $\mathbb{Q}$ fest lassen, können jeweils zu homomorphen* Einbettungen σ_i *von $\mathbb{Q}(\rho)$ in $\mathbb{C}$ fortgesetzt werden. Ist $\rho_i \in \mathbb{R}$, so nennen wir σ_i eine* reellwertige Einbettung*, sonst* komplexwertige Einbettung*.*

Basierend auf diesen Einbettungen können wir algebraische Zahlen als sogenannte *Konjugiertenvektoren* auffassen. Dabei entspricht einer Zahl α das Tupel $\underline{\alpha} = (\sigma_1(\alpha), \ldots, \sigma_n(\alpha))^T \in \mathbb{C}^n$. Die Abbildung $\underline{\cdot} : \mathbb{Q}(\rho) \longrightarrow \mathbb{C}^n$, $x \longmapsto \underline{x}$ ist ein injektiver Ringhomomorphismus, wenn die Operationen auf den Vektoren aus $\mathbb{C}^n$ komponentenweise definiert werden.

Diese Darstellung enthält eine gewisse Redundanz: Da die Minimalpolynome algebraischer Zahlen nur Koeffizienten aus $\mathbb{Q}$ enthalten, ist mit jeder komplexen Zahl auch die konjugiert–komplexe Zahl eine Nullstelle des Minimalpolynoms. Wenn wir also die Einbettungen geeignet sortieren, so erhalten wir folgenden Aufbau des Konjugiertenvektors:

$$\underline{\alpha} = \left(\sigma_1(\alpha), \ldots, \sigma_r(\alpha), \sigma_{r+1}(\alpha), \ldots, \sigma_{r+\frac{n-r}{2}}(\alpha), \overline{\sigma_{r+1}(\alpha)}, \ldots, \overline{\sigma_{r+\frac{n-r}{2}}(\alpha)}\right)^T,$$

wobei $\sigma_1, \ldots, \sigma_r$ reellwertige Einbettungen und $\sigma_{r+1}, \ldots, \sigma_{r+\frac{n-r}{2}}$ komplexwertige Einbettungen sind und mit $\overline{x}$ das Konjugiert–komplexe von x bezeichnet wird.

Damit können wir also unsere Repräsentation etwas vereinfachen, indem wir die konjugiert–komplexen Einbettungen weglassen. Damit erhalten wir:

$$\begin{aligned} \underline{\alpha} \;=\; & \Big(\sigma_1(\alpha), \ldots, \sigma_r(\alpha), \Re(\sigma_{r+1}(\alpha)), \Im(\sigma_{r+1}(\alpha)), \\ & \ldots, \Re\left(\sigma_{r+\frac{n-r}{2}}(\alpha)\right), \Im\left(\sigma_{r+\frac{n-r}{2}}(\alpha)\right)\Big)^T \in \mathbb{R}^n, \end{aligned} \tag{1.1}$$

wobei $\Re(x)$ den Realteil einer komplexen Zahl x bezeichnet und $\Im(x)$ ihren Imaginärteil. Hierbei ist zu beachten, dass bei Berechnungen mit diesen Vektoren die zuvor komponentenweise erfolgende Multiplikation über $\mathbb{C}$ jetzt durch eine etwas kompliziertere Multiplikation über $\mathbb{R}$ ersetzt wird.

Im weiteren nennen wir $\underline{\alpha}$ (in der vereinfachten Repräsentation) den *Konjugiertenvektor* einer algebraischen Zahl α, und wir definieren:

1.1.4. Definition
Sei K ein algebraischer Zahlkörper. Dann definieren wir den Ringmonomorphismus „$\underline{\cdot}$“ wie folgt:

$$\begin{aligned}\underline{\cdot}: \quad & K \longrightarrow \mathbb{R}^n \quad , \\ & \alpha \longmapsto \underline{\alpha} = \Big(\sigma_1(\alpha), \ldots, \sigma_r(\alpha), \Re(\sigma_{r+1}(\alpha)), \Im(\sigma_{r+1}(\alpha)), \\ & \qquad \ldots, \Re\left(\sigma_{r+\frac{n-r}{2}}(\alpha)\right), \Im\left(\sigma_{r+\frac{n-r}{2}}(\alpha)\right)\Big)^T,\end{aligned}$$

wobei die Addition auf $\mathbb{R}^n$ komponentenweise definiert ist und die Multiplikation wie folgt:

- *Die ersten r Komponenten werden komponentenweise multipliziert.*
- *Die folgenden $n-r$ Komponenten werden im Sinne der Multiplikation in $\mathbb{C}$ paarweise miteinander multipliziert.*

Diese Darstellung erlaubt es insbesondere, den Begriff der euklidischen Länge auf algebraische Zahlen anzuwenden und z.B. Reduktionsverfahren wie den LLL–Algorithmus zu benutzen. Später werden wir in der Tat nutzbringende Anwendungen dieser Technik sehen.

Später benutzen wir auch noch folgende Begriffe:

1.1.5. Definition
Sei ρ eine algebraische Zahl mit $[\mathbb{Q}(\rho) : \mathbb{Q}] = n$ und seien $\sigma_1, \ldots, \sigma_n$ die n Einbettungen von $\mathbb{Q}(\rho)$ in $\mathbb{C}$. Für ein Element $\alpha \in \mathbb{Q}(\rho)$ bezeichnen wir dann mit $\mathcal{N}(\alpha) := \prod_{i=1}^{n} \sigma_i(\alpha)$ die Norm *von α (in $\mathbb{Q}(\rho)$). Das Polynom $c_\alpha(x) := \prod_{i=1}^{n}(x - \sigma_i(\alpha))$ nennen wir* charakteristisches Polynom *von α.*

Schließlich wollen wir den Begriff der Norm noch im folgenden Sinne erweitern:

1.1.6. Definition
Sei K ein algebraischer Zahlkörper mit mit $[K:\mathbb{Q}]=n$ und seien $\sigma_1,\ldots,\sigma_n$ die n Einbettungen von K in $\mathbb{C}$. Für Elemente $\alpha_1,\ldots,\alpha_m\in K$ und Unbestimmte $x_1,\ldots,x_m$ ($m\in\mathbb{N}$) bezeichnen wir dann das Polynom $\mathcal{N}(x_1\alpha_1+\cdots+x_m\alpha_m):=\prod_{i=1}^{n}\sigma_i(x_1\alpha_1+\cdots+x_m\alpha_m)\in\mathbb{Q}[x_1,\ldots,x_m]$ als Normform *von $\alpha_1,\ldots,\alpha_m$ (in K).*

Eine Auswertung dieses Ausdrucks zeigt unmittelbar, dass eine solche Normform ein homogenes Polynom vom Grad n in m Variablen ist. Die Tatsache, dass die Koeffizienten in $\mathbb{Q}$ liegen, ergibt sich dabei als Folgerung aus der Galoistheorie. Sind $\alpha_1,\ldots,\alpha_m$ algebraisch ganze Zahlen, so liegen diese Koeffizienten sogar in $\mathbb{Z}$.

1.2 Ordnungen und Einheiten

1.2.1 Definitionen

Ebenso wie die (rationalen) ganzen Zahlen in $\mathbb{Q}$ bilden auch die ganzen Zahlen eines algebraischen Zahlkörpers einen Ring, der von ähnlich großer Bedeutung wie $\mathbb{Z}$ ist.

1.2.1. Definition
Ein 1–Oberring $\mathcal{O}\subset K:=\mathbb{Q}(\rho)$ von $\mathbb{Z}$, der zugleich freier $\mathbb{Z}$–Modul maximalen Rangs aus algebraisch ganzen Zahlen ist, heißt Ordnung *des algebraischen Zahlkörpers K. Die eindeutig bestimmte (bzgl. Inklusion) maximale Ordnung heißt* Maximalordnung*. Diese ist dadurch charakterisiert, dass sie alle algebraisch ganzen Elemente des Körpers enthält und heißt daher oft auch* Ganzheitsring*. Sie wird im folgenden mit $\mathcal{O}_K$ bezeichnet.*

Im Gegensatz zu $\mathbb{Z}$ ist $\mathcal{O}_K$ jedoch kein Hauptidealring, es gilt noch nicht einmal der Satz von der eindeutigen Primzahlzerlegung (z.B. in $\mathbb{Z}[\sqrt{-5}]$ gilt: $6=2\cdot 3=(1+\sqrt{-5})(1-\sqrt{-5})$). Damit lässt sich die Teilbarkeitstheorie, die für $\mathbb{Z}$ von so großer Bedeutung ist, zunächst nicht auf algebraische Zahlkörper bzw. genauer auf deren Ganzheitsringe übertragen.

Falls ρ eine algebraisch ganze Zahl ist, ist $\mathbb{Z}[\rho]$ selbst eine Ordnung. Diese wird oft als *Gleichungsordnung* bezeichnet und wird implizit durch das Körperpolynom gegeben. Aus dieser Gleichungsordnung lässt sich durch sukzessives

Maximieren die Maximalordnung errechnen, wie wir im nächsten Kapitel sehen werden.

Ein Maß für die Größe einer Ordnung ist die Diskriminante der Ordnung.

1.2.2. Definition (Diskriminante)
Für eine $\mathbb{Q}$–Basis $\alpha_1, \ldots, \alpha_n$ von K und komplexe Einbettungen $\sigma_1, \ldots, \sigma_n$ von K in $\mathbb{C}$ bezeichnen wir mit

$$\Delta(\alpha_1, \ldots, \alpha_n) := \left[\det \begin{pmatrix} \sigma_1(\alpha_1) & \ldots & \sigma_1(\alpha_n) \\ \vdots & \ddots & \vdots \\ \sigma_n(\alpha_1) & \ldots & \sigma_n(\alpha_n) \end{pmatrix} \right]^2$$

die Diskriminante *von $\alpha_1, \ldots, \alpha_n$.*

1.2.3. Bemerkung
Die Diskriminante hat folgende Eigenschaften:

- Sie ist offensichtlich unabhängig von der Reihenfolge der α_i und σ_j.
- Offensichtlich ist sie invariant unter unimodularen Transformationen der Basis.
- Es ist stets $\Delta(\alpha_1, \ldots, \alpha_n) \in \mathbb{Q}$.

Aufgrund der Invarianz unter unimodularen Transformationen der Basis ist die folgende Definition sinnvoll:

1.2.4. Definition
Unter der Diskriminanten $\Delta_{\mathcal{O}}$ einer Ordnung $\mathcal{O}$ verstehen wir die Diskriminante einer beliebigen $\mathbb{Z}$–Basis der Ordnung. Als Körperdiskriminante Δ_K eines Körpers K bezeichnen wir die Diskriminante der maximalen Ordnung, d.h. des Ganzheitsringes $\mathcal{O}_K$.

Der zentrale Satz, auf dem die Berechnung des Ganzheitsringes aufbaut, ist der folgende:

1.2.5. Satz
Sind $\mathcal{O}$ und $\mathcal{Q}$ zwei Ordnungen mit $\mathcal{O} \subset \mathcal{Q}$, so gilt:

$$\Delta_{\mathcal{O}} = \Delta_{\mathcal{Q}} [\mathcal{Q} : \mathcal{O}]^2,$$

wobei $[\mathcal{Q} : \mathcal{O}]$ den Index (im gruppentheoretischen Sinne) von $\mathcal{O}$ in $\mathcal{Q}$ bezeichnet. Falls $\mathcal{Q}$ die Maximalordnung ist, nennt man $[\mathcal{Q} : \mathcal{O}]$ auch den Index von $\mathcal{O}$.

Eine etwas allgemeinere Version dieses Satzes findet sich in [Neu92], Satz 2.12, S. 16, die hier angegebene ist jedoch für unsere Zwecke ausreichend.

Für praktische Berechnungen von besonderer Bedeutung sind die sogenannten Einheiten:

1.2.6. Definition
Ein Element $\alpha \in \mathcal{O}$ heißt Einheit (in $\mathcal{O}$), falls $\alpha^{-1} \in \mathcal{O}$. Die Menge aller Einheiten in $\mathcal{O}$ wird mit $\mathcal{O}^*$ bezeichnet.

Die Einheiten sind dadurch gekennzeichnet, dass die Norm einer Einheit betragsmäßig gerade 1 ist (vgl. [Neu92]).

Der zentrale Satz über die Einheiten der Maximalordnung stammt von Dirichlet und lautet:

1.2.7. Satz (Dirichlet'scher Einheitensatz)
Die Einheitengruppe $\mathcal{O}_K^*$ von $\mathcal{O}_K$ ist das direkte Produkt der endlichen zyklischen Gruppe $\mu(K)$ der in K gelegenen Einheitswurzeln und einer freien abelschen Gruppe vom Rang $r+s-1$, wobei r wiederum die Zahl der reellwertigen Einbettungen von K und $2s$ die Zahl komplexwertiger Einbettungen von K beschreibt.

Zum Beweis vergleiche [Neu92], Kapitel I, Paragraph 7 (S. 41–45).

1.2.8. Definition
Die Erzeuger der freien abelschen Gruppe aus dem vorangegangenen Satz heißen Fundamentaleinheiten *von K.*

Die Einheitswurzeln sind leicht zu behandeln, problematisch sind nur die restlichen Einheiten. Um diese in den Griff zu bekommen, betrachten wir die Konjugiertenvektoren und wenden den Logarithmus auf die Komponenten an. Dadurch reduzieren wir die multiplikative Gruppe dieser Einheiten auf ein Gitter in $\mathbb{R}^r \times \mathbb{C}^s$, so dass wir die bekannten Gitteralgorithmen anwenden können.

Genauer benutzen wir folgende Abbildung:

$$\begin{aligned} \mathrm{Log} : K^* &\longrightarrow \mathbb{R}^{r+s-1} \\ \alpha &\longmapsto (\log|\sigma_1(\alpha)|, \ldots, \log|\sigma_r(\alpha)|, \\ &\qquad \log(\sqrt{2}|\sigma_{r+1}(\alpha)|), \ldots, \log(\sqrt{2}|\sigma_{r+s-1}(\alpha)|)\Big) \end{aligned}$$

1.2.9. Definition
Seien $\epsilon_1, \ldots, \epsilon_{r+s-1}$ die Fundamentaleinheiten eines algebraischen Zahlkörpers

K. Die Determinante des von $\mathrm{Log}(\epsilon_1), \ldots, \mathrm{Log}(\epsilon_{r+s-1})$ *erzeugten Gitters heißt* Regulator $R(K)$ *von* K.

Bei der Berechnung der Fundamentaleinheiten sind sogenannte *unabhängige Einheiten* von Bedeutung, d.h. Einheiten, die die Eigenschaft haben, dass sich 1 nicht als Potenzprodukt der Form $1 = \prod_{i=1}^{k} \epsilon_i^{e_i}$ mit mindestens einem von 0 verschiedenen e_i darstellen lässt. Ein System von $r+s-1$ unabhängigen Einheiten erzeugt dann stets eine Untergruppe der von den Fundamentaleinheiten erzeugten Gruppe, die endlichen Index hat. Dies hat insbesondere zur Folge, dass für irgendwelche unabhängigen Einheiten $\epsilon_1, \ldots, \epsilon_{r+s-1}$ die Determinante des von $\mathrm{Log}(\epsilon_1), \ldots, \mathrm{Log}(\epsilon_{r+s-1})$ erzeugten Gitters stets ein ganzzahliges Vielfaches des Regulators ist.

1.3 Moduln und Ideale

1.3.1 Definitionen

1.3.1. Definition
Sei K ein algebraischer Zahlkörper vom Grad n. Ein $\mathbb{Z}$–Modul $M \subset K$ heißt vollständig, *wenn er n (über $\mathbb{Q}$) linear unabhängige Elemente enthält. Eine Menge $\{\alpha_1, \ldots, \alpha_m\} \subset M$ heißt* Basis *von M, wenn $\alpha_1, \ldots, \alpha_m$ linear unabhängig sind und den Modul erzeugen, d.h. wenn $M = \{c_1\alpha_1 + \cdots + c_m\alpha_m, c_1, \ldots, c_m \in \mathbb{Z}\}$.*

Üblicherweise werden die folgenden Operationen für Moduln definiert:

1.3.2. Definition
Seien M_1 und M_2 zwei Moduln. Dann ist

$$\begin{aligned} M_1 + M_2 &:= \{m_1 + m_2 | m_1 \in M_1, m_2 \in M_2\} \text{ (Summe)} \\ M_1M_2 := M_1 \cdot M_2 &:= \{\sum_{(m_1,m_2)\in S} m_1 \cdot m_2 | S \subseteq M_1 \times M_2 \text{ endlich}\} \text{ (Produkt)} \\ M_1 \cap M_2 &:= \{m | m \in M_1 \text{ und } m \in M_2\} \text{ (Durchschnitt)} \\ (M_1 : M_2) &:= \{x \in K | x \cdot M_2 \subseteq M_1\} \text{ (Pseudo – Division).} \end{aligned}$$

Ein wichtiger Spezialfall des Moduls ist das Ideal.

1.3.3. Definition
Sei $\mathcal{O}$ eine Ordnung eines algebraischen Zahlkörpers K. Ein Modul $\mathfrak{a}$ mit $\mathfrak{a} \cdot \mathcal{O} =$

$\mathfrak{a}$ *heißt* (gebrochenes) Ideal *in* $\mathcal{O}$ *(kurz:* $\mathcal{O}$*–Ideal). Ist* $\mathfrak{a} \subset \mathcal{O}$*, so heißt* $\mathfrak{a}$ *ganzes Ideal.*

Ist $\mathcal{O}$ *der Ganzheitsring, so sagt man auch* $\mathfrak{a}$ *ist Ideal von* K*.*

Ist zudem $\mathfrak{a} = \alpha\mathcal{O}$ *für ein* $\alpha \in K$*, so heißt* $\mathfrak{a}$ Hauptideal.

Die ganzen Ideale sind gerade die Ideale im üblichen Sinne. Für die gebrochenen Ideale $\mathfrak{a}$ gilt, dass es eine ganze Zahl $n \in \mathbb{Z} \setminus \{0\}$ gibt, so dass $n \cdot \mathfrak{a}$ ein ganzes Ideal ist. Das kleinste positive n mit dieser Eigenschaft heißt *Nenner des Ideals* $\mathfrak{a}$.

1.3.4. Definition
Eine Zahl $\alpha \in K$ *heißt* Multiplikator *des vollständigen Moduls* M*, falls* $\alpha M \subset M$*. Die Menge aller Multiplikatoren ist ein Ring mit* 1 *und heißt* Multiplikatorenring*, in Zeichen* $\mathcal{O}_M$*.*

Es gelten die folgenden Eigenschaften:

- Jeder Multiplikatorenring ist zugleich eine Ordnung.
- Jeder Modul ist Ideal seines Multiplikatorenrings.
- Summe, Produkt und Durchschnitt von Idealen sind wiederum Ideale.

Ist $\mathfrak{a}$ ein $\mathcal{O}$–Ideal, so dass das Ergebnis der Pseudo–Division $(\mathcal{O} : \mathfrak{a})$ wieder ein Ideal ist, so heißt $\mathfrak{a}$ *invertierbar*. Es ist dann $\mathfrak{a}^{-1} := (\mathcal{O} : \mathfrak{a})$ und $\mathfrak{a} \cdot \mathfrak{a}^{-1} = \mathcal{O}$.

In der Maximalordnung ist jedes von (0) verschiedene Ideal invertierbar. Insbesondere bilden die Ideale $\neq (0)$ eine multiplikative Gruppe $\mathcal{I}(K)$, in der die von (0) verschiedenen Hauptideale eine Untergruppe $\mathcal{H}(K)$ bilden.

1.3.5. Definition
Sei $\mathcal{I}(K)$ *die Gruppe der von* (0) *verschiedenen (gebrochenen) Ideale eines algebraischen Zahlkörpers* K *und* $\mathcal{H}(K)$ *die Untergruppe der Hauptideale. Dann heißt die Restklassengruppe* $Cl(K) := \mathcal{I}(K)/\mathcal{H}(K)$ *die* Klassengruppe *von* K*,* $h(K) := |Cl(K)|$ *heißt die* Klassenzahl *von* K*.*

Ist die Klassenzahl 1, so ist die Maximalordnung ein Hauptidealring und die Teilbarkeitstheorie von $\mathbb{Z}$ lässt sich sofort übertragen. Andernfalls führt ein Umweg über die Ideale zum Ziel.

In Analogie zu $\mathbb{Z}$ betrachten wir zunächst die ganzen Ideale.

1.3.6. Definition
Seien $\mathfrak{a}$ *und* $\mathfrak{b}$ *zwei ganze Ideale. Wir sagen* $\mathfrak{a}$ teilt $\mathfrak{b}$*, in Zeichen* $\mathfrak{a} \mid \mathfrak{b}$*, wenn* $\mathfrak{b}$ *in* $\mathfrak{a}$ *enthalten ist.*

In diesem Falle gilt in Analogie zu den ganzen Zahlen, dass $(\mathfrak{b} : \mathfrak{a})$ wieder ein ganzes Ideal ist. Damit lässt sich nun die Teilbarkeitstheorie, wie wir sie aus $\mathbb{Z}$ kennen, leicht auf Ideale übertragen. Wir erhalten folgende Begriffe:

1.3.7. Definition

- *Ein ganzes Ideal* $\mathfrak{m}$ *heißt* maximal*, falls das einzige ganze Ideal, das* $\mathfrak{m}$ *echt umfasst,* $\mathcal{O}$ *selbst ist (dies entspricht den unzerlegbaren Zahlen in* $\mathbb{Z}$*).*
- *Ein ganzes Ideal* $\mathfrak{p}$ *heißt* prim*, falls für je zwei ganze Ideale* $\mathfrak{a}$ *und* $\mathfrak{b}$ *aus* $\mathfrak{p} \mid \mathfrak{a} \cdot \mathfrak{b}$ *folgt, dass entweder* $\mathfrak{p} \mid \mathfrak{a}$ *oder* $\mathfrak{p} \mid \mathfrak{b}$ *gilt.*

Man sieht nun leicht, dass die Summe zweier Ideale, so wie wir sie oben definiert haben, gerade der größte gemeinsame Teiler der beiden Ideale ist und dass der Durchschnitt gerade dem kleinsten gemeinsamen Vielfachen entspricht.

In Analogie zu $\mathbb{Z}$ gilt auch hier die Äquivalenz von unzerlegbaren Idealen und Primidealen. Schließlich gilt der folgende Hauptsatz, der das Analogon zur eindeutigen Primzahlzerlegung in $\mathbb{Z}$ darstellt:

1.3.8. Satz
Ist $\mathfrak{a}$ *ein ganzes Ideal, so existieren eindeutig (bis auf die Reihenfolge) bestimmte, paarweise verschiedene Primideale* $\mathfrak{p}_1, \ldots \mathfrak{p}_n$*, so dass*

$$\mathfrak{a} = \prod_{i=1}^{n} \mathfrak{p}_i^{e_i}, e_i \in \mathbb{N}.$$

Diese eindeutige Zerlegbarkeit überträgt sich in evidenter Weise auch auf gebrochene Ideale.

Ebenso wie für algebraische Zahlen gibt es auch für Ideale wieder den Begriff der Norm.

1.3.9. Definition
Die Norm $\mathcal{N}(\mathfrak{a})$ *eines Ideals* $\mathfrak{a}$ *ist der Betrag der Determinante der Transformationsmatrix* $T \in \mathbb{Z}^{n \times n}$*, die eine Basis der Ordnung zu einer Basis des Ideals transformiert.*

Für ein ganzes Ideal $\mathfrak{a}$ ist die Norm gerade die Anzahl der Elemente des Quotientenrings $\mathcal{O}/\mathfrak{a}$, und für Hauptideale $(\alpha) := \alpha\mathcal{O}$ ergibt sich $\mathcal{N}((\alpha)) = |\mathcal{N}(\alpha)|$, d.h. die Idealnorm ist mit der Elementnorm verträglich. Darüber hinaus ist die Norm multiplikativ, wenn man sie auf die Menge der invertierbaren Ideale einschränkt. Insbesondere gilt der folgende Satz:

1.3.10. Satz
Sei K ein algebraischer Zahlkörper. Dann gilt für die Ideale von K: Die Abbildung

$$\begin{aligned} \mathcal{N} : \mathcal{I}(K) &\longrightarrow \mathbb{Q}, \\ \mathfrak{a} &\longmapsto \mathcal{N}(\mathfrak{a}) \end{aligned}$$

ist ein Homomorphismus.

Eine Folgerung hieraus ist, dass die Norm eines Primideals stets die Potenz einer Primzahl ist, so dass es eine eindeutig bestimmte Primzahl gibt, zu der ein Primideal gehört, man sagt, das Primideal *liegt über der Primzahl*. Insbesondere gilt:

1.3.11. Satz
Ist p eine Primzahl, so gibt es in einem algebraischen Zahlkörper K vom Grad n Primideale $\mathfrak{p}_1, \ldots, \mathfrak{p}_k$ $(k \leq n)$, die über p liegen, und es ist

$$(p) = p\mathcal{O}_K = \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}.$$

Dabei ist $\mathcal{N}(\mathfrak{p}_i) = p^{f_i}$ für $1 \leq i \leq k$ mit

$$\sum_{i=1}^{k} e_i f_i = n.$$

Zum Beweis vergleiche [Neu92], Satz 8.2, S. 48f.

1.3.12. Definition
Seien p, k und $\mathfrak{p}_i, e_i, f_i (1 \leq i \leq k)$ wie im vorangehenden Satz. Dann heißt e_i Verzweigungsgrad *und f_i heißt* Trägheitsgrad *von $\mathfrak{p}_i$.*

Damit haben wir nun alle Ingredienzien, um die zentrale Formel für die Berechnung der Klassenzahl und der Klassengruppe formulieren zu können:

1.3.13. Satz (Klassenzahlformel)
Sei K ein algebraischer Zahlkörper mit Klassenzahl h, Regulator R, Diskriminante Δ_K, w Einheitswurzeln und r reellen und $2s$ komplexen Einbettungen. Dann gilt

$$\frac{2^r(2\pi)^s}{w\sqrt{|\Delta_K|}}hR = \prod_{p\in\mathbb{P}} \frac{1-\frac{1}{p}}{\prod_{\mathfrak{p}\text{über } p \text{ prim}} 1-\frac{1}{\mathcal{N}(\mathfrak{p})}}$$

Zum Beweis vergleiche [Neu92], Kapitel VII, Theorem 5.9 und Korollar 5.11.

Die Bedeutung dieser Formel liegt darin, dass sie einen Zusammenhang zwischen Regulator und Klassenzahl einerseits und einem numerisch approximierbaren Ausdruck andererseits herstellt. Damit ist es möglich, eine Vermutung über Klassenzahl und Regulator innerhalb der Grenzen der Rechengenauigkeit zu verifizieren.

Unsere Methoden zur Klassenzahl– und Regulatorberechnung beruhen darauf, dass man ein Vielfaches von hR (bzw. eine Approximation daran) bestimmt. Mit dieser analytischen Formel ist uns nun das Werkzeug in die Hand gegeben, um zu prüfen, ob wir hR gefunden haben. Falls dabei eine solche Prüfung zeigt, dass wir nur ein Vielfaches gefunden haben, so können wir eine Abschätzung geben, um welchen Faktor wir höchstens noch neben dem Resultat liegen.

Im Falle imaginärquadratischer Zahlkörper weiß man sogar, dass $R = 1$ gilt, und kann diese Formel theoretisch unmittelbar benutzen, um die Klassenzahl zu berechnen. In der Praxis zeigt sich allerdings, dass man typischerweise eine extrem hohe Rechengenauigkeit verwenden muss, um das korrekte Ergebnis zu erhalten. Da zudem die zu berechnende Reihe ein recht schlechtes Konvergenzverhalten hat, ist dies für Zahlkörper mit großer Diskriminante kein praktikables Verfahren.

1.4 LiDIA

Zum Abschluss dieser Einleitung wollen wir darauf hinweisen, dass wir die Algorithmen, die wir beschreiben werden, in der Bibliothek LiDIA implementiert haben.

Bei LiDIA handelt es sich um eine Klassenbibliothek für algebraische Zahlentheorie, die seit 1994 am Lehrstuhl von Professor Buchmann zunächst in Saarbrücken und seit 1996 in Darmstadt entwickelt wird.

Dabei verfolgen wir das Konzept, die besonders zeitkritischen Teile wie

die Langzahlarithmetik und die Speicherverwaltung, den sogenannten Kern, zu kapseln und durch ein einheitliches Interface für den Rest der Bibliothek zur Verfügung zu stellen. Dadurch wird es leicht möglich den Kern auszutauschen und so bessere Langzahlarithmetiken und Speicherverwaltungen zu nutzen. So hoffen wir, auch längerfristig die Verwendbarkeit der Bibliothek bei geringem Wartungsaufwand zu gewährleisten.

Mittels der Funktionen des Interfaces wurden weitere Arithmetiken, etwa für $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{C}$ und $\mathbb{Z}/m\mathbb{Z}$ implementiert, sowie Klassen für Polynome, Vektoren und Matrizen über diesen Bereichen. Daneben werden Algorithmen für Gitter (etwa zur Gitterbasisreduktion), zum Faktorisieren von ganzen Zahlen und Polynomen und zur Berechnung diskreter Logarithmen in endlichen Primkörpern angeboten.

Unter Benutzung fast aller zur Verfügung stehenden Strukturen haben wir eine Arithmetik für algebraische Zahlkörper implementiert, die neben den elementaren Operationen für Zahlen und Ideale auch die komplizierteren Algorithmen realisiert, die wir im Weiteren beschreiben werden. Dabei existiert eine spezialisierte Variante für quadratische Zahlkörper, da sich für diesen Fall viele Probleme vereinfachen. So ist es für quadratische Zahlkörper gegebener Diskriminante z.B. ein triviales Problem, eine Ganzheitsbasis anzugeben.

Schließlich gibt es ein ausführliches Referenzhandbuch, das auch interaktiv zur Verfügung steht.

Die Gesamtstruktur der Bibliothek ist in Abbildung 1.1 skizziert.

Als Implementierungssprache für die Bibliothek haben wir dabei C++ gewählt, einerseits da diese Sprache ebenso wie C eine effiziente, maschinennahe Programmierung unterstützt, andererseits da die Klassenstruktur eine Abbildung mathematischer Strukturen auf die Bibliotheksstruktur erleichtert.

Im folgenden wollen wir uns nun auf die Beschreibung der Klassen für Polynome und für allgemeine algebraische Zahlkörper konzentrieren. Für die Beschreibung der anderen Klassen, die im Laufe der Berechnungen verwendet werden, verweisen wir auf das Manual ([98]) sowie auf die Arbeiten von Thomas Papanikolaou ([Pap97]), Patrick Theobald ([The96]), Thomas Pfahler([Pfa98]), Susanne Wetzel ([Wet98]) und Werner Backes([Bac98]).

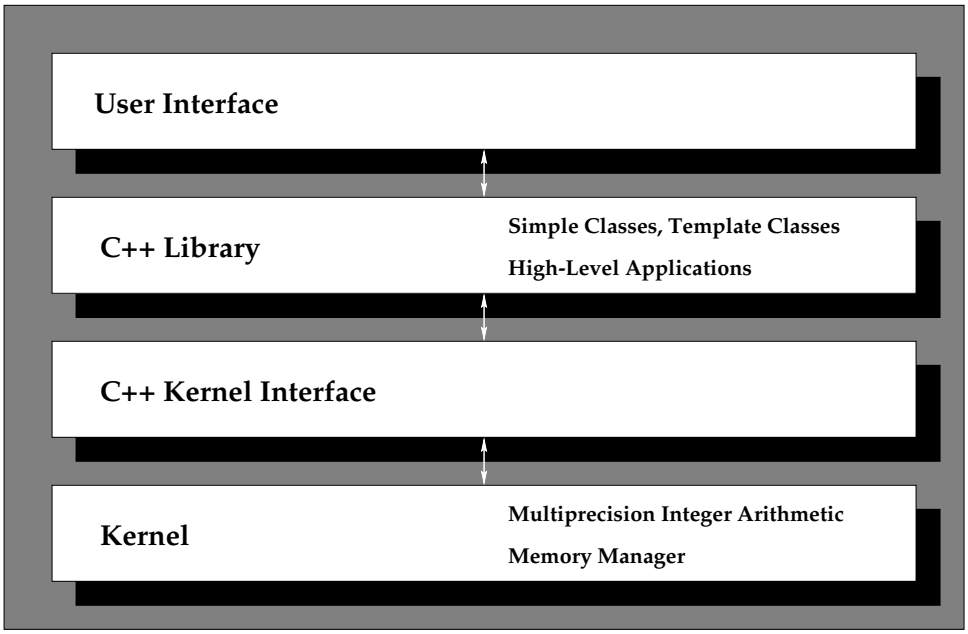

Abbildung 1.1: Die Struktur der Klassenbibliothek LiDIA.

Kapitel 2

Zahlkörper, Ordnungen und ihre Elemente

In diesem Kapitel wollen wir uns der Darstellung der Zahlkörper und Ordnungen widmen, in denen alle weiteren Berechnungen spielen werden. Daneben werden wir auch zeigen, wie wir algebraische Zahlen darstellen. Außerdem wollen wir einige grundlegende Algorithmen und ihre Implementierung diskutieren.

Die grundlegende Information, die für alle Klassen, die mit algebraischen Zahlkörpern zu tun haben, von Bedeutung ist, ist dabei eine $\mathbb{Q}$–Basis des Zahlkörpers bzw. allgemeiner die Information darüber, wie zwei algebraische Zahlen zu multiplizieren sind. Daher wollen wir uns zunächst der Klasse `nf_base` zuwenden, die diese grundlegenden Informationen beinhaltet und verwaltet. Dabei wird sich die Bedeutung der Polynomarithmetik zeigen, so dass wir anschließend einen Abschnitt über diese einschieben werden.

Danach zeigen wir, wie man mit Hilfe dieser Klassen die eigentlich interessanten Objekte der Mathematik darstellen kann. Dabei beginnen wir mit algebraischen Zahlen und wenden uns dann den abstrakteren Begriffen des Zahlkörpers und der Ordnung zu. Im nächsten Kapitel wollen wir dann auf Moduln und Ideale eingehen.

2.1 Die Klasse `nf_base`

Die Klasse `nf_base` ist aus mathematischer Sicht ein künstliches Gebilde, daher wollen wir zunächst die Existenzberechtigung dieser Klasse erläutern. Danach wollen wir sehen, welche Informationen die Klasse beinhaltet und wie sie in LiDIA benutzt wird. Dabei werden wir sehen, dass die Implementierung so ge-

staltet ist, dass diese Klasse vom Anwender niemals explizit verwendet werden muss. Statt dessen können Objekte von Klassen verwendet werden, die direkte Entsprechungen mathematischer Strukturen sind.

Speziell bei Zahlkörpern und ihren Ganzheitsringen ist der Sprachgebrauch in der Mathematik etwas nachlässig, häufig spricht ein Mathematiker vom Zahlkörper, wenn er implizit seinen Ganzheitsring meint, etwa wenn von der Klassenzahl eines Zahlkörpers die Rede ist. Auch algebraische Zahlen und (gebrochene) Ideale werden häufig wahlweise bezüglich einer Ordnung oder bezüglich eines Zahlkörpers gebraucht. Dabei liefert der Zahlkörper bzw. die Ordnung für das praktische Rechnen mit algebraischen Zahlen im wesentlichen die Information darüber, wie Zahlen multipliziert werden.

Daher bietet es sich an, diese Kerninformation in einer eigenen Klasse, die wir `nf_base` nennen, zu kapseln und Zahlkörper und Ordnungen als syntaktischen Zucker mit evtl. Zusatzinformationen darüber zu streuen. Der Zugriff soll dabei wahlweise über die Ordnung oder den Zahlkörper erfolgen, damit wir den Sprachgebrauch der Mathematik so natürlich wie möglich abbilden können.

Darüber hinaus haben wir die Objekte dieser Klasse mit einer dynamischen Lebensdauer versehen, so dass es uns nicht passieren kann, dass wir zwei Variablen zur Darstellung algebraischer Zahlen Werte in einem Zahlkörper zuweisen, mit diesen aber später nicht mehr rechnen können, weil die Informationen darüber, wie man diese Zahlen multipliziert, bereits wieder gelöscht wurden. Da diese Informationen einen beträchtlichen Platzbedarf haben können, ist es nämlich nicht ohne weiteres möglich, sie bei jedem Objekt mit abzuspeichern.

Nach dieser Rechtfertigung wollen wir uns nun der Realisierung dieser Klasse zuwenden. Die abgespeicherten Informationen der Klasse `nf_base` ergeben sich im wesentlichen aus den verschiedenen Möglichkeiten einen Zahlkörper oder eine Ordnung darzustellen, die wir nun diskutieren werden.

In der Praxis geht es zunächst darum, Elemente von Zahlkörpern oder Ordnungen addieren, subtrahieren, multiplizieren und dividieren zu können. Sei dazu $\omega_1, \ldots, \omega_n$ eine fixierte Basis eines Körpers bzw. einer Ordnung und seien $\alpha = \sum_{i=1}^{n} a_i\omega_i$ und $\beta = \sum_{i=1}^{n} b_i\omega_i$ zwei Zahlen in diesem Körper. Dann sind

$$\alpha \pm \beta = \sum_{i=1}^{n} (a_i \pm b_i)\omega_i \tag{2.1}$$

und

$$\alpha \cdot \beta = \sum_{i=1}^{n}\sum_{j=1}^{n} (a_i b_j)(\omega_i\omega_j). \tag{2.2}$$

Gleichung (2.1) zeigt, dass Addition und Subtraktion wie in jedem Vektorraum komponentenweise erklärt sind. Für die Multiplikation ist es nach (2.2) notwendig und hinreichend, zu erklären, wie je zwei Basiselemente miteinander multipliziert werden. Dies erweist sich auch für alle anderen Operationen in der Tat als ausreichend.

Insbesondere auf die Division wollen wir später eingehen, es sei hier nur gesagt, dass sie sich auf die Multiplikation und das Lösen eines Gleichungssystems zurückführen lässt.

Darstellung mittels eines Polynoms

Zunächst können wir Zahlkörper und *manche* Ordnungen durch ein normiertes irreduzibles Polynom $f(x)$ über $\mathbb{Z}$ beschreiben. Ist dieses Polynom vom Grad n, so bildet $(1, x, x^2, \ldots, x^{n-1})$ zugleich eine $\mathbb{Z}$–Basis der sogenannten *Gleichungsordnung* und eine $\mathbb{Q}$–Basis des Zahlkörpers. Die Multiplikation bzgl. dieser Basis kann sehr leicht realisiert werden, da die Basiselemente x^i und x^j gerade entsprechend der Polynomarithmetik zu $x^{i+j} mod(f)$ multipliziert werden. Algebraische Zahlen entsprechen hier gerade Polynomen in $\mathbb{Q}[x]$ vom Grad kleiner n und die Multiplikation entspricht einer Polynommultiplikation mit anschließender Reduktion modulo des Körperpolynoms. Für zwei algebraische Zahlen $\alpha = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ und $\beta = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}$ gilt also:

$$\alpha \cdot \beta = (a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}) \cdot (b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}) \bmod (f) \quad (2.3)$$

Um diese Darstellung zu unterstützen, bietet die Klasse `nf_base` die Komponente `f` vom Typ `polynomial < bigint >` an.

2.1.1. Beispiel

Wir betrachten als fortlaufendes Beispiel den Körper $K := \mathbb{Q}(\sqrt{-2} + \sqrt{5})$. Dieser wird durch das Polynom $f = x^4 - 6 * x^2 + 49$ beschrieben. □

Allgemeiner kann man zusätzlich eine Transformationsmatrix $T \in \mathrm{GL}(n, \mathbb{Q})$ angeben, so dass für die verwendete Basis $(\omega_1, \ldots, \omega_n)$ gilt:

$$(\omega_1, \ldots, \omega_n) = (1, x, x^2, \ldots, x^{n-1}) \cdot T.$$

In dieser Art lässt sich insbesondere eine $\mathbb{Z}$–Basis für *jede* Ordnung angeben. Zieht man den Hauptnenner der Einträge in der Transformationsmatrix nach vorne, so enthält die Matrix selbst nur noch ganzzahlige Einträge. Die Multiplikation wird dann etwas komplizierter, zwei Zahlen $\alpha = a_1\omega_1 + \cdots + a_n\omega_n$ und

$\beta = b_1\omega_1 + \cdots + b_n\omega_n$ werden nun wie folgt multipliziert: Zunächst transformieren wir mittels T die beiden Zahlen auf eine Darstellung bezüglich der Basis $(1, x, x^2, \ldots, x^{n-1})$, diese multiplizieren wir wie schon gesehen und erhalten ein modulo f reduziertes Polynom:

$$\begin{aligned}\gamma' &= c_0 + c_1 x + \cdots + c_{n-1}x^{n-1} \\ &= \left((1, x, x^2, \ldots, x^{n-1}) \cdot T \cdot (a_1, \ldots, a_n)^T\right) \cdot \\ &\quad \left((1, x, x^2, \ldots, x^{n-1}) \cdot T \cdot (b_1, \ldots, b_n)^T\right). \end{aligned} \tag{2.4}$$

Das gesuchte Produkt von α und β bzgl. der gewünschten Basis erhalten wir dann als

$$\gamma = (\omega_1, \ldots, \omega_n) \cdot T^{-1} \cdot (c_0, \ldots, c_{n-1})^T. \tag{2.5}$$

Diese Darstellung wird mit den Komponenten `den` vom Typ `bigint` und `base` vom Typ `matrix<bigint>` unterstützt, die den Hauptnenner und die entsprechende ganzzahlige Matrix enthalten. Dabei benutzen wir die Konvention, dass wir die Einheitsmatrix kürzer durch eine leere Matrix darstellen. Dadurch erreichen wir einen besonders einfachen Test darauf, ob bereits $(1, x, x^2, \ldots, x^{n-1})$ die Basis ist. Da dieser Fall in unseren Routinen stets eine Sonderbehandlung erfährt, sparen wir mit diesem kleinen Trick Rechenzeit und Speicherplatz.

2.1.2. Beispiel

Wenn wir in unserem Beispiel etwa mit der Gleichungsordnung $\mathcal{O} := \mathbb{Z}[\sqrt{-2} + \sqrt{5}]$ starten, dann ist $\Delta_{\mathcal{O}} = 20070400 = 2^{14} \cdot 5^2 \cdot 7^2$. Bei der Berechnung der Maximalordnung mit den Methoden, die wir am Ende des Kapitels beschreiben werden, zeigt sich dann, dass gilt: $[\mathcal{O}_K : \mathcal{O}] = 112 = 2^4 \cdot 7$, also ergibt sich $\Delta_K = 1600$. Dabei wird $\mathcal{O}_K$ als $\mathbb{Z}$–Modul von den Elementen $\omega_1 = 1$, $\omega_2 = (\sqrt{-2} + \sqrt{5})$, $\omega_3 = 3/4 + 1/2(\sqrt{-2} + \sqrt{5}) + 1/4(\sqrt{-2} + \sqrt{5})^2$ und $\omega_4 = 1/4 + 1/28(\sqrt{-2} + \sqrt{5}) + 1/4(\sqrt{-2} + \sqrt{5})^2 + 1/28(\sqrt{-2} + \sqrt{5})^3)$ erzeugt, kann also mittels `den`$= 28$ und

$$\texttt{base} = \begin{pmatrix} 28 & 0 & 21 & 7 \\ 0 & 28 & 14 & 1 \\ 0 & 0 & 7 & 7 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

dargestellt werden. □

Diese Repräsentation ist dann sehr gut geeignet, wenn man in der Gleichungsordnung rechnet. Vermutlich bewährt sich diese Darstellung in der Pra-

xis auch für Transformationsmatrizen mit „kleinen“ Einträgen, dies haben wir allerdings nicht weiter untersucht. In der praktischen Anwendung haben wir nämlich das Problem, dass wir neben der Matrix T selbst auch die Matrix T^{-1} benötigen, die schlimmstenfalls Einträge der n-fachen Länge der originalen Matrix hat, so dass die theoretischen Laufzeit- und Platzabschätzungen zumindest nicht besser sind, als für die zweite Darstellungsmethode, der wir uns sogleich zuwenden wollen. Der Vorteil dieser Darstellung liegt darin, dass es Algorithmen gibt, deren Effizienz wesentlich gesteigert werden kann, wenn das Körperpolynom bekannt ist: Im Falle der Maximalordnungsberechnung kann man etwa den Dedekind–Test anwenden, wenn man das Körperpolynom kennt (vgl. Satz 2.5.4), und im Falle der Zerlegung von Primzahlen in Primideale kann man die Faktorisierung des Körperpolynoms modulo einer Primzahl benutzen (vgl. Satz 4.2.1).

Darstellung mittels der Multiplikationstafel

Eine alternative Darstellungsmethode, die stärker die Multiplikation von zwei Basiselementen in den Vordergrund stellt, beruht darauf, eine sogenannte Multiplikationstafel zu speichern. In (2.2) müssen wir nur wissen, was das Produkt von je zwei Basiselementen ist. Daher berechnen wir Zahlen w_{ijk} aus $\mathbb{Z}$ mit

$$\omega_i \omega_j = \sum_{k=1}^{n} w_{ijk} \omega_k. \tag{2.6}$$

Diese Werte speichern wir in einer Tabelle, der sogenannten *Multiplikationstafel*, mit deren Hilfe wir nun multiplizieren können, (2.2) vereinfacht sich dann zu

$$\alpha \cdot \beta = \sum_{k=1}^{n} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} a_i b_j w_{ijk} \right) \omega_k. \tag{2.7}$$

Die Multiplikationstafel kann man wie in (2.3) bzw. (2.4) und (2.5) gesehen aus der vorangegangenen Darstellung einer Basis konstruieren, Wenn wir es mit einer Ordnung zu tun haben, muss das Produkt zweier Elemente wieder eine $\mathbb{Z}-$Linearkombination der Basisvektoren ergeben, daher sind in diesem Fall die Einträge dieser Tabelle ganze Zahlen. Der Einfachheit halber verlangen wir, dass auch jeder Zahlkörper durch eine Basis dargestellt wird, die in der Multiplikationstafel die Verwendung ganzer Zahlen erlaubt. Diese Multiplikationstafel ist offensichtlich abhängig von der Wahl der Basis, daher definieren wir:

2.1.3. Definition
Sei $\Omega = (\omega_1, \ldots, \omega_n)$ *eine* $\mathbb{Z}$*–Basis einer Ordnung* $\mathcal{O}$*. Dann bezeichnen wir mit* $\mathrm{MT}(\Omega) := (w_{ijk})_{1 \le i,j,k \le n}$ *gemäß (2.6) die* Multiplikationstafel *zu dieser Basis. Mit* $|\mathrm{MT}(\Omega)|$ *bezeichnen wir den Betrag des betragsgrößten Eintrags in der Multiplikationstafel.*

Schließlich ist es auch möglich, aus solch einer Multiplikationstafel ein irreduzibles Polynom zu konstruieren, so dass die Gleichungsordnung, die zu diesem Polynom gehört und die Ordnung, zu der die Multiplikationstabelle gehört, denselben Quotientenkörper haben. Dies ist in [Nei94], Kapitel 4 ausführlich beschrieben.

Die Multiplikationstafel wird in der Komponente `table` der Klasse `nf_base` gespeichert. Hierbei beachte man, dass nur die w_{ijk} mit $i \ge j$ gespeichert werden, da aufgrund der Kommutativität der Multiplikation $w_{ijk} = w_{jik}$ gilt. Dabei legen wir willkürlich fest, dass w_{ijk} an Position $(\frac{i(i-1)}{2} + j - 1, k - 1)$ der Variablen `table` vom Typ `matrix <bigint>` gespeichert wird.

In unserer Implementierung können wahlweise entweder nur die Multiplikationstafel oder nur das Polynom (und optional die Transformationsmatrix) initialisiert und verwendet werden. Mittels der Funktionen `compute_base()` bzw. `compute_table()` wird bei Bedarf die jeweils andere Darstellung berechnet. Falls der verwendete C++ Compiler das ANSI–Schlüsselwort `mutable` unterstützt, wird diese neu berechnete Information jeweils auch abgespeichert, falls das Schlüsselwort noch nicht unterstützt wird, wird in Funktionen, die die betreffende Variable nicht ändern dürfen, jeweils eine lokale Kopie der Variable erzeugt, und die Berechnung darauf ausgeführt. Eine solche Funktion ist z.B. der Ausgabeoperator, da wir stets eine Darstellung mittels Polynom und Transformationsmatrix berechnen und ausgeben.

Vergleich der beiden Darstellungen

Wir wollen die Unterschiede und Gemeinsamkeiten der beiden Darstellungsmethoden zusammenfassen:

- Die Repräsentation mittels Polynom und Transformationsmatrix besteht aus wenigen Zahlen (n, falls wir keine Transformationsmatrix speichern müssen, $n^2 + n + 1$ sonst). Die Darstellung mit einer Multiplikationstafel benötigt dagegen $\frac{1}{2}n^3 + \frac{1}{2}n^2$ ganze Zahlen.

- Die Größe der Einträge ist in beiden Fällen stark davon abhängig, welche Basis einer Ordnung bzw. eines Körpers ausgewählt wurde.

- Die Addition von zwei Zahlen ist in beiden Fällen gleichermaßen einfach, denn hierfür genügt ein Koeffizientenvektor, die Basis wird nicht wirklich benötigt.

- Die Multiplikation von Zahlen aus der Ordnung ist ebenfalls in beiden Fällen relativ leicht. In der Darstellung mittels einer Multiplikationstafel kann man einfach (2.7) anwenden, in der Darstellung mittels Polynom und Transformationsmatrix benötigt man allerdings neben der Polynomarithmetik in (2.3) bzw. (2.4) vor allem die Inverse der Transformationsmatrix. Diese muss man entweder jedesmal ausrechnen oder aber abspeichern. Da die Einträge in der inversen Matrix jedoch nur mit der Hadamardschranke abgeschätzt werden können, kann deren Darstellung um einen Faktor in der Größenordnung des Körpergrads mehr Platz benötigen, als die Einträge in der Ausgangsmatrix. Damit macht man mit dem Abspeichern aber den Vorteil der Speicherplatzersparnis evtl. wieder zunichte, und auch der Zeitaufwand steigt unter Umständen bis auf das Niveau der Darstellung mittels einer Multiplikationstafel, da zwar weniger arithmetische Operationen zur Multiplikation benötigt werden, aber evtl. längere Zahlen beteiligt sind.

- Ein Test auf Gleichheit ist in der Darstellung mittels Polynom und Transformationsmatrix sehr schwierig, da es viele unterschiedliche Darstellungen dieser Art für ein– und dieselbe Basis geben kann, die jeweils mit verschiedenen Polynomen beginnen. Hingegen gibt es eine eineindeutige Beziehung zwischen Basen und Multiplikationstafeln, so dass in diesem Falle der Vergleich sehr einfach ist. Allerdings lässt sich auch auf diese Weise nicht testen, ob zwei Basen den gleichen $\mathbb{Z}$–Modul erzeugen. Dies ist nur für den Sonderfall einer Darstellung mit Polynom und Transformationsmatrix, bei der „zufällig“ für beide Basen das gleiche Polynom Verwendung findet, einigermaßen effizient möglich.

Darstellung mittels Konjugiertenvektoren

Neben einer Darstellung der Basisvektoren mit einer Transformationsmatrix relativ zur Basis $(1, x, x^2, \ldots, x^{n-1})$ ist auch eine Darstellung mittels Konjugier-

tenvektoren möglich. Dabei betten wir die algebraischen Zahlen homomorph nach $\mathbb{C}^n$ ein, wobei x auf den Vektor bestehend aus allen komplexen Nullstellen des Minimalpolynoms abgebildet wird. Die Operationen sind dann komponentenweise möglich. Eine etwas detailliertere Abhandlung dieser Darstellung findet sich z.B. in [Coh95], Kapitel 4.2.4.

2.1.4. Beispiel
Für die Zahl $\alpha = 3 + (\sqrt{-2} + \sqrt{5}) \in \mathbb{Q}(\sqrt{-2} + \sqrt{5})$ erhalten wir den Konjugiertenvektor $\underline{\alpha} = (3 + \sqrt{5} + \sqrt{2}i, 3 + \sqrt{5} - \sqrt{2}i, 3 - \sqrt{5} + \sqrt{2}i, 3 - \sqrt{5} - \sqrt{2}i) \in \mathbb{C}^4$. Wenn wir auf die Redundanz verzichten, die sich durch die Aufzählung von je zwei konjugiert–komplexen Werten ergibt, schreiben wir kürzer: $\underline{\alpha} = (3 + \sqrt{5}, \sqrt{2}, 3 - \sqrt{5}, -\sqrt{2}) \in \mathbb{R}^4$. □

Der große Nachteil dieser Darstellung besteht in den damit verbundenen Präzisionsproblemen: Um auch für Zahlen, die sich als Linearkombination mit großen Koeffizienten aus den Basiselementen ergeben, korrekte Berechnungen anstellen zu können, ist eine sehr hohe Präzision bei der Berechnung der Konjugiertenvektoren nötig – damit werden die Berechnungen für Zahlen mit kleinen Koeffizienten aber ineffizient.

Daher bietet die Klasse `nf_base` *nicht* die Möglichkeit, die Konjugiertenvektoren als Alternative zu einer der beiden anderen Darstellungen zu benutzten. Mittels der Konjugiertenvektoren ist es jedoch möglich, eine Basis aus „kurzen“ Elementen zu bestimmen (etwa mittels LLL–Reduktion). Dies erweist sich in der Praxis häufig als nützlich, manchmal sogar als notwendig – etwa um eine kurze Darstellung einer Ordnung zu berechnen, vgl. [Nei94]. Daher bietet unsere Implementierung die Möglichkeit, in der Komponente `conjugates` vom Typ `bigfloat_matrix` *zusätzlich* die Konjugiertenvektoren zu speichern. Diese werden, sobald sie zum ersten Mal benötigt werden, mittels der Funktion `compute_conjugates` berechnet. Diese Funktion berechnet zunächst die komplexen Nullstellen des Körperpolynoms, welches im Bedarfsfall mittels `compute_base` bestimmt werden kann. Diese Nullstellen werden in willkürlicher Reihenfolge angeordnet, wobei allerdings erst alle reellen Nullstellen aufgezählt werden und dann jeweils ein Vertreter eines jeden Paares aus komplexer und konjugiert-komplexer Nullstelle, so dass wir einen Vektor der Form (1.1) erhalten, also in der Form

$$\begin{aligned}\underline{\alpha} \;=\; & (\sigma_1(\alpha), \ldots, \sigma_r(\alpha), \\ & \Re(\sigma_{r+1}(\alpha)), \Im(\sigma_{r+1}(\alpha)), \ldots, \Re(\sigma_{r+\frac{n-r}{2}}(\alpha)), \Im(\sigma_{r+\frac{n-r}{2}}(\alpha)))^T \in \mathbb{R}^n,\end{aligned}$$

Damit ist nun die Reihenfolge der Einbettungen festgelegt und wir können die Konjugiertenvektoren der Potenzen von α bestimmen. Dann können wir die Transformationsmatrix `base` anwenden, um die Konjugiertenvektoren der gewünschten Basis des Körpers zu erhalten.

Damit sieht die Klasse `nf_base` nun wie folgt aus:

```
class nf_base{
  // The classes allowed to change bases get
  // immediate access. They need to do all the
  // work, since otherwise reference counting gets
  // difficult. So everything is private, since
  // nobody is allowed to explicitly use this class!

  friend class number_field; // Those need access to all
  friend class order;        // information of class nf_base.

  friend class alg_number;   // Those need access only to
  friend class module;       // the variable "current_base".
  friend class alg_ideal;
  friend class prime_ideal;

  // Additional friend functions:
  friend bool operator ==(const number_field &,
                          const number_field &);
  friend ostream& operator<<(ostream&, const number_field&);
  friend istream& operator>>(istream&, number_field&);
  friend const bigint & disc(const order & O); // discriminant
  friend ostream& operator<<(ostream &, const order &);
  friend istream& operator>>(istream &, order &);
  friend void multiply(alg_number &, const alg_number &,
                       const alg_number &);
  friend bigint_matrix rep_matrix(const alg_number &);
  friend istream& operator>>(istream &, alg_number &);
  friend istream& operator>>(istream &, module &);

  static nf_base * current_base;  // Default nf_base
  static nf_base * dummy_base;    // Dummy nf_base
```

```
int references;                 // How many objects
                                // use this base.
mutable polynomial <bigint> f;   // the irreducible polynomial
mutable lidia_size_t real_roots; // number of real roots

mutable bigint_matrix base;
mutable bigint den;
mutable math_matrix <bigint> table; // multiplication table

// Internal information, that is frequently needed
// and therefore stored with the order, as soon,
// as it is computed:
mutable math_vector <bigint> One; // Representation of 1
bigfloat_matrix  conjugates;    // column i contains conjugates
                                // of w_i (0 <= i < degree().
// Functions for checking, what is already computed:
bool table_computed() const;
void compute_table() const;

bool base_computed() const;
void compute_base() const;

void compute_conjugates();

// whether to use MT or polynomial for multiplying.
bool using_necessary() const;

const bigfloat & get_conjugate(lidia_size_t, lidia_size_t);
const bigfloat_matrix & get_conjugates();

const polynomial <bigint> & which_polynomial() const;
lidia_size_t degree() const;

// Constructor and Destructor
nf_base():references(0), f(), real_roots(0), base(),
  den(1), table(), One(), conjugates() {}     // Do nothing.
```

```
  ~nf_base(){}          // Just call member destructors.

  //  Default copy constructor and assignment are OK.

  void assign(const nf_base&);
  // Input and output are used by number_field and order!
  friend ostream&
  operator<<(ostream&, const nf_base&); // read/write the
  friend istream&
   operator>>(istream&, nf_base&);      // polynomial/trafo

public:
  const math_vector <bigint> & get_one() const;
  void dec_ref();
  void inc_ref();
};
```

Besondere Beachtung verdient die Tatsache, dass diese Klasse fast vollkommen vom Anwender isoliert ist, und insbesondere Konstruktor und Destruktor nur von den Konstruktoren bzw. Destruktoren der Klassen `order` und `number_field` aufgerufen werden. Die Ein- und Ausgabe ist vollständig in der Klasse `nf_base` konzentriert und die Klassen `order` und `number_field` rufen diese Funktionen lediglich auf.

2.2 Polynomarithmetik in LiDIA

Wir haben bereits gesehen, dass die Polynomarithmetik eine wichtige Grundlage für das Rechnen in algebraischen Zahlkörpern bildet. In erster Linie benötigen wir dabei Polynome über $\mathbb{Z}$. Mit diesen lassen sich einerseits algebraische Zahlkörper darstellen. Daneben kann man sie aber auch zur Repräsentation von Zahlen aus einer Gleichungsordnung benutzen bzw. für algebraische Zahlen allgemein, wenn man einen evtl. Nenner getrennt behandelt.

Zur Berechnung der komplexen Einbettungen (Definition 1.1.3) eines algebraischen Zahlkörpers müssen wir natürlich komplexe Nullstellen eines Polynoms berechnen können. Dabei bietet es sich an, diese Algorithmen gleich in voller Allgemeinheit für Polynome mit komplexen Koeffizienten zu implementieren.

Im Vorgriff auf die spätere Verwendung, weisen wir darauf hin, dass wir zur Berechnung von Primidealen (Satz 4.2.1) ganzzahlige Polynome auch modulo einer Primzahl p faktorisieren müssen.

Im folgenden wollen wir nun unsere Implementierung beschreiben. Dabei stellen wir die Klassenhierarchie von der Basisklasse `base_polynomial < T >` über die Klasse `field_polynomial < T >` bis hin zur allgemeinen Interface-Klasse `polynomial < T >` dar und gehen dann insbesondere auf die Spezialisierungen von `polynomial < T >` für `bigint` und `bigcomplex` ein.

Um die Wiederverwendbarkeit zu gewährleisten, bot es sich an, Polynome in einer Vererbungshierarchie von Template–Klassen zu realisieren. Dabei gibt es zunächst eine Klasse `base_polynomial`, die die Basisoperationen für Polynome, also Addition, Subtraktion, Multiplikation und Zugriff auf die Koeffizienten zur Verfügung stellt. Die Datenstruktur und der Funktionsumfang dieser Klasse werden durch den folgenden Code beschrieben. Dabei ist `coeff` das Array von Koeffizienten, `coeff[i]` repräsentiert den Koeffizienten von x^i und `deg` ist der Grad des Polynoms. Das Nullpolynom stellen wir mit Grad -1 und einem Nullpointer für `coeff` dar.

```
template < class T > class base_polynomial
{
protected:
  /**
   ** base_polynomial<T> a = c[0] + c[1]*X + ... + c[deg]*X^deg
   **/

  T * coeff;
  lidia_size_t deg;

  void copy_data(T * d, const T * vd, lidia_size_t al);

public:

  /**
   ** constructors and destructor
   **/

  base_polynomial();
```

```
base_polynomial(const T & x);
base_polynomial(const T * v, lidia_size_t d);
base_polynomial(const base_vector < T > v);
base_polynomial(const base_polynomial < T > &p);
~base_polynomial();

/**
 ** friend functions
 **/

/**  comparisons  **/

friend bool operator == (const base_polynomial < T > &a,
                         const base_polynomial < T > &b);

friend bool operator != (const base_polynomial < T > &a,
                         const base_polynomial < T > &b);

T lead_coeff() const;
friend T lead_coeff(const base_polynomial < T > &a);

T const_term() const;
friend T const_term(const base_polynomial < T > &a);

/**
 ** member functions
 **/

void remove_leading_zeros();

int set_data ( const T * d, lidia_size_t l );
T* get_data () const;

lidia_size_t degree() const;

void set_degree(lidia_size_t);
```

```
bool is_zero() const;
bool is_one() const;
bool is_x() const;

/**
 ** assignment
 **/

void assign(const  T &);
base_polynomial < T > &
operator = (const T &a);
void assign(const base_polynomial < T > &);
base_polynomial < T > &
operator = (const base_polynomial < T > &a);

void assign_zero();
void assign_one();
void assign_x();

friend void swap(base_polynomial < T > &a,
                 base_polynomial < T > &b);

/**
 ** operator overloading
 **/

T & operator[] (lidia_size_t i) const;
T operator() (const T & value) const;

/**
 ** arithmetic procedures
 **/

friend void negate(base_polynomial < T > & c,
                   const base_polynomial < T > &a);
```

```
friend void add(base_polynomial < T > & c,
                const base_polynomial < T > & a,
                const base_polynomial < T > & b);
friend void add(base_polynomial < T > & c,
                const base_polynomial < T > & a,
                const T & b);
friend void add(base_polynomial < T > & c,
                const T & b,
                const base_polynomial < T > & a);
friend void subtract(base_polynomial < T > & c,
                     const base_polynomial < T > & a,
                     const base_polynomial < T > & b);
friend void subtract(base_polynomial < T > & c,
                     const base_polynomial < T > & a,
                     const T & b);
friend void subtract(base_polynomial < T > & c,
                     const T & b,
                     const base_polynomial < T > & a);
friend void multiply(base_polynomial < T > & c,
                     const base_polynomial < T > & a,
                     const base_polynomial < T > & b);
friend void multiply(base_polynomial < T > & c,
                     const base_polynomial < T > & a,
                     const T & b);
friend void multiply(base_polynomial < T > & c,
                     const T & b,
                     const base_polynomial < T > & a);
friend void power(base_polynomial < T > & c,
                  const base_polynomial < T > & a,
                  const bigint & b);

/**
 ** operator overloading
 **/

friend base_polynomial < T >
```

```
operator - (const base_polynomial < T > &a);
friend base_polynomial < T >
operator + (const base_polynomial < T > &a,
            const base_polynomial < T > &b);
friend base_polynomial < T >
operator + (const base_polynomial < T > & a,
            const T & b);
friend base_polynomial < T >
operator + (const T & b,
            const base_polynomial < T > & a);
friend base_polynomial < T >
operator - (const base_polynomial < T > &a,
            const base_polynomial < T > &b);
friend base_polynomial < T >
operator - (const base_polynomial < T > &a,
            const T &b);
friend base_polynomial < T >
operator - (const T &a,
            const base_polynomial < T > &b);
friend base_polynomial < T >
operator * (const base_polynomial < T > &a,
            const base_polynomial < T > &b);
friend base_polynomial < T >
operator * (const base_polynomial < T > &a,
            const T &b);
friend base_polynomial < T >
operator * (const T &b,
            const base_polynomial < T > &a);
base_polynomial < T > &
operator += (const base_polynomial < T > &a);
base_polynomial < T > &
operator += (const T &a);
base_polynomial < T > &
operator -= (const base_polynomial < T > &a);
base_polynomial < T > &
operator -= (const T &a);
```

```
  base_polynomial < T > &
  operator *= (const base_polynomial < T > &a);
  base_polynomial < T > &
  operator *= (const T &a);

  /**
   ** functions
   **/
  friend void derivative(base_polynomial < T > &c,
                         const base_polynomial < T > &a);
  friend base_polynomial < T >
  derivative(const base_polynomial < T > & a);

  /**
   ** input / output
   **/

  friend istream & operator >> (istream &s,
                                base_polynomial < T > &c);
  istream & read_verbose(istream &);
  friend ostream & operator << (ostream &s,
                                const base_polynomial < T > &a);
};
```

Die Funktionsnamen in diesem Codefragment sind im wesentlichen selbsterklärend, wie es der Philosophie von LiDIA entspricht. Dabei sind, was den Funktionsumfang betrifft, die Funktionen `negate`, `add`, `subtract` und `multiply` eigentlich überflüssig, ebenso wie in anderen LiDIA–Klassen dienen sie allerdings zur Vermeidung unnötiger Kopieraktionen, die z.B. bei der Addition mehr als 50 Prozent der Laufzeit ausmachen können, wenn man statt `add(c,a,b)` die Version `c=a+b` verwendet.

Da in unserem Kontext typischerweise Polynome von kleinem Grad auftreten, ist es hinreichend, zur Implementierung der Funktionen die Standardalgorithmen zu verwenden, da sich z.B. Algorithmen zur schnellen Multiplikation, wie sie etwa in [CK91] beschrieben werden, erst für vergleichsweise großen Grad lohnen.

Zusätzlich haben wir von dieser Klasse die Klasse `field_polynomial` abge-

leitet, die Operationen wie Division und symbolische Integration zur Verfügung stellt, die nur über Körpern wohldefiniert sind. Diese ist wie folgt realisiert:

```
template < class T >
class field_polynomial: public base_polynomial < T >
{
public:

  /**
   ** constructors and destructor
   **/

  field_polynomial():
    base_polynomial < T > (){}
  field_polynomial(T x):
    base_polynomial < T > (x){}
  field_polynomial(const T * v, lidia_size_t d):
    base_polynomial < T > (v,d){}
  field_polynomial(const base_vector < T > v):
    base_polynomial < T > (v){}{}
  field_polynomial(const base_polynomial < T > &p):
    base_polynomial < T > (p){}
  ~field_polynomial(){}

  field_polynomial < T > &
  operator = (const base_polynomial < T > &a);

  /**
   ** Division and related stuff
   **/

  friend void div_rem(field_polynomial < T > &,
                      field_polynomial < T > &,
                      const base_polynomial < T > &,
                      const base_polynomial < T > &);
  friend void divide(field_polynomial < T > & c,
                     const base_polynomial < T > & a,
```

```
                    const T & b);
  friend void divide(field_polynomial < T > & q,
                    const base_polynomial < T > & a,
                    const base_polynomial < T > & b);
  friend void remainder(field_polynomial < T > & r,
                       const base_polynomial < T > & a,
                       const base_polynomial < T > & b);
  friend void power_mod(field_polynomial < T > & c,
                       const base_polynomial < T > & a,
                       const bigint & b,
                       const base_polynomial < T > & f);

  field_polynomial < T > &
  operator /= (const base_polynomial < T > &a);
  field_polynomial < T > &
  operator /= (const T &a);
  field_polynomial < T > &
  operator %= (const base_polynomial < T > &a);


  friend void integral(field_polynomial < T > &c,
                       const base_polynomial < T > &a);
  friend void gcd(field_polynomial < T > &d,
                  const base_polynomial < T > &aa,
                  const base_polynomial < T > &bb);
  friend void xgcd(field_polynomial < T > &d,
                   field_polynomial < T > &x,
                   field_polynomial < T > &y,
                   const base_polynomial < T > &aa,
                   const base_polynomial < T > &bb);
};
```

Die Datenstruktur bleibt im Vergleich zu `base_polynomial` somit also unverändert, Konstruktoren und Destruktoren rufen lediglich die entsprechenden Methoden von `base_polynomial` auf, und die Funktionalität wird etwas erweitert. Dabei fällt auf, dass der Divisionsoperator nicht implementiert ist und das einige Funktionen, die ein `field_polynomial` zurückgeben sollten, feh-

len. Dies ist darauf zurückzuführen, dass dieser Operator sinnvollerweise ein `field_polynomial` zurückgeben sollte, gleichzeitig aber für Polynome über den ganzen Zahlen dieser Operator sinnvollerweise mit Pseudo–Division verknüpft wird, die aber offensichtlich kein `field_polynomial<bigint>` zurückgeben kann, da die ganzen Zahlen keinen Körper bilden. Dabei treten bei verschiedenen Compilern unterschiedliche Fehler zu Tage, die dazu führen, dass die Version des Codes, die von `gcc-2.7.x` übersetzt werden kann, von `Visual C++ 5.0` als fehlerhaft zurückgewiesen wird und umgekehrt. Bei den restlichen fehlenden Funktionen tritt dasselbe Problem mit den Rückgabetypen auf. Diese Einschränkungen sind jedoch relativ harmlos, wie wir sogleich sehen werden.

Entsprechend der mathematischen Notation, wo man einfach $R[X]$ für den Polynomring mit Koeffizienten aus R schreibt, bieten wir ein zusätzliches Interface zu den Polynomklassen, das die Unterscheidung nach `base_polynomial` und `field_polynomial` automatisch vornimmt. Damit kann der Benutzer einfach `polynomial<T>` schreiben. Die Klasse `polynomial` erbt einfach von der Klasse `base_polynomial` und bietet zunächst keine zusätzliche Funktionalität. Allerdings gibt es für die in unserem Kontext relevanten Typen geeignete Spezialisierungen dieser Klasse, die dann von `field_polynomial` statt `base_polynomial` erben und/oder zusätzliche Operationen anbieten. So ist z.B. `polynomial<bigint>` eine Spezialisierung von `base_polynomial<bigint>`, die zusätzlich Funktionen für die Faktorisierung modulo Primzahlen sowie Routinen, die auf Pseudo–Division basieren (Division mit Rest und ggT bzw. erweiterter ggT), zur Verfügung stellt. Daneben bietet diese Spezialisierung auch Cast–Operatoren an, die ein Polynom über $\mathbb{Z}$ in ein Polynom über $\mathbb{Q}$, $\mathbb{R}$ oder $\mathbb{C}$ konvertieren. Dadurch wird es möglich, ohne weitere Komplikationen, genau wie man es aus der Mathematik gewohnt ist, ein Polynom mit ganzzahligen Koeffizienten mit einem Polynom mit reellen Koeffizienten zu multiplizieren oder die komplexen Nullstellen eines solchen Polynoms zu berechnen. Im folgenden geben wir nun die Deklaration der Klasse `polynomial<bigint>` wieder:

```
class polynomial <bigint>: public base_polynomial <bigint>
{
public:

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

```
polynomial():
  base_polynomial <bigint> (){}
polynomial(bigint x):
  base_polynomial <bigint> (x){}
polynomial(const bigint * v, lidia_size_t d):
  base_polynomial <bigint> (v,d){}
polynomial(const base_vector <bigint> v):
  base_polynomial <bigint> (v){}
polynomial(const base_polynomial <bigint> &p):
  base_polynomial <bigint> (p){}
polynomial(const polynomial <bigint> &p):
  base_polynomial <bigint> (p){}
~polynomial(){}

polynomial <bigint> &
operator = (const base_polynomial <bigint> &a);
polynomial <bigint> &
operator = (const polynomial <bigint> &a);

/**
 ** Cast operators:
 **/

operator base_polynomial<bigrational>() const;
operator base_polynomial<bigfloat>() const;
operator base_polynomial<bigcomplex>() const;

/**
 ** Pseudo - division and related stuff
 **/

friend void div_rem(polynomial <bigint> &q,
                    polynomial <bigint> &r,
                    const base_polynomial <bigint> &a,
                    const base_polynomial <bigint> &b);
friend void divide(polynomial <bigint> & c,
```

```
                    const base_polynomial <bigint> & a,
                    const bigint & b);
friend void divide(polynomial <bigint> & q,
                    const base_polynomial <bigint> & a,
                    const base_polynomial <bigint> & b);
friend void remainder(polynomial <bigint> & r,
                       const base_polynomial <bigint> & a,
                       const base_polynomial <bigint> & b);
friend void remainder(polynomial <bigint> & c,
                       const base_polynomial <bigint> & a,
                       const bigint & b);
friend void power_mod(polynomial <bigint> & c,
                       const base_polynomial <bigint> & a,
                       const bigint & b,
                       const base_polynomial <bigint> & f);
friend polynomial <bigint>
operator / (const base_polynomial <bigint> &a,
            const base_polynomial <bigint> &b);
friend polynomial <bigint>
operator / (const base_polynomial <bigint> &a,
            const bigint & b);
friend polynomial <bigint>
operator % (const base_polynomial <bigint> &a,
            const base_polynomial <bigint> &b);
friend polynomial <bigint>
operator % (const base_polynomial <bigint> &a,
            const bigint &b);
polynomial <bigint> &
operator /= (const base_polynomial <bigint> &a);
polynomial <bigint> &
operator /= (const bigint &a);
polynomial <bigint> &
operator %= (const bigint &b);
polynomial <bigint> &
operator %= (const base_polynomial <bigint> &a);
```

```
  /**
   ** Gcd's and related stuff, i.e. content and primitive part.
   **/

  friend bigint cont(const base_polynomial <bigint> &a);
  friend polynomial <bigint>
  pp(const base_polynomial <bigint> &a);
  friend polynomial <bigint>
  gcd(const base_polynomial <bigint> &aa,
      const base_polynomial <bigint> &bb);
  friend polynomial <bigint>
  xgcd(polynomial <bigint> &x,
       polynomial <bigint> &y,
       const base_polynomial <bigint> &aa,
       const base_polynomial <bigint> &bb);
  /**
   ** Number of real roots
   **/

  friend lidia_size_t
  no_of_real_roots(const base_polynomial<bigint>& poly_T);
};

/**
 ** Resultant and Discriminant
 **/

bigint resultant(const polynomial <bigint> &aa,
                 const polynomial <bigint> &bb);
bigint discriminant(const polynomial <bigint> &a);
```

Für die Koeffizientenbereiche $\mathbb{Q}$ und $\mathbb{R}$ und $\mathbb{C}$ bietet die Klasse `polynomial` jeweils Spezialisierungen der Klasse `field_polynomial` sowie Cast–Operatoren. Für $\mathbb{C}$ gibt es insbesondere die Möglichkeit komplexe Nullstellen eines Polynoms zu berechnen. Der Algorithmus zur Nullstellenberechnung basiert auf der Newton–Iteration. Die Implementierung folgt im wesentlichen der Beschreibung in [Coh95], Kapitel 3.6.3. Wegen der großen Ähnlichkeit dieser drei Spezia-

lisierungen, wollen wir exemplarisch nur die interessanteste, nämlich die für bigcomplex herausgreifen.

```
class polynomial <bigcomplex>:
public field_polynomial <bigcomplex>
{
public:
  polynomial():
    field_polynomial <bigcomplex> (){}
  polynomial(bigcomplex x):
    field_polynomial <bigcomplex> (x){}
  polynomial(const bigcomplex * v, lidia_size_t d):
    field_polynomial <bigcomplex> (v,d){}
  polynomial(const base_vector <bigcomplex> v):
    field_polynomial <bigcomplex>(v){}
  polynomial(const base_polynomial <bigcomplex> &p):
    field_polynomial <bigcomplex> (p){}
  ~polynomial(){}

  polynomial <bigcomplex> &
  operator = (const base_polynomial <bigcomplex> &a);
  friend bigcomplex root(const base_polynomial <bigcomplex> &,
                         const bigcomplex &);
  friend void cohen(const base_polynomial <bigcomplex> &,
                    bigcomplex *, int, int &);
  friend void roots(const base_polynomial <bigcomplex> &,
                    bigcomplex *);
  friend bigcomplex
  integral(const bigfloat &, const bigfloat &,
           const base_polynomial <bigcomplex> &);
  friend polynomial < bigcomplex >
  operator / (const base_polynomial < bigcomplex > &a,
              const base_polynomial < bigcomplex > &b);
  friend polynomial < bigcomplex >
  operator / (const base_polynomial < bigcomplex > &a,
              const bigcomplex & b);
  friend polynomial < bigcomplex >
```

```
  operator % (const base_polynomial < bigcomplex > &a,
              const base_polynomial < bigcomplex > &b);

  friend polynomial < bigcomplex >
  integral(const base_polynomial < bigcomplex > & a);

  friend polynomial < bigcomplex >
  gcd(const base_polynomial < bigcomplex > &aa,
      const base_polynomial < bigcomplex > &bb);

  friend polynomial < bigcomplex >
  xgcd(polynomial < bigcomplex > &x,
       polynomial < bigcomplex > &y,
       const base_polynomial < bigcomplex > &aa,
       const base_polynomial < bigcomplex > &bb);
};
```

An diesem Codefragment erkennt man insbesondere, dass hier nun die Funktionen realisiert wurden, die wir eben in der Klasse `field_polynomial` vermisst haben. Ihre Implementierung besteht dabei jeweils aus einer einzeiligen Inline–Funktion, allerdings müssen aufgrund der fehlerhaften Compiler diese wenigen Zeilen nun in jeder Spezialisierung hinzugefügt werden, statt diese nur einmal für die Klasse `field_polynomial` schreiben zu können. Nun können z.B. einheitlich alle Divisionsoperatoren, insbesondere sowohl derjenige für `polynomial<bigint>` als auch derjenige für `polynomial<bigcomplex>` ein `polynomial` zurückgeben, so dass das ursprüngliche Compiler–Problem nicht mehr auftritt.

Zusammenfassend ergibt sich für die Polynome die Vererbungsstruktur aus Abbildung 2.1, in der ein Pfeil von einer Klasse A zu einer Klasse B anzeigt, dass Klasse A von Klasse B abgeleitet ist.

Neben den damit zur Verfügung gestellten Algorithmen benötigen wir zur Zerlegung von Primzahlen in Primideale und für den Dedekind–Test auch noch Algorithmen zur Zerlegung von Polynomen über $\mathbb{F}_p$ in irreduzible Faktoren. Diese wurden von Thomas Pfahler in der Klasse `Fp_polynomial` implementiert (siehe [Pfa98]). Dabei wurden die Algorithmen und Faktorisierungsstrategien zwar vorwiegend für große Primzahlen und große Polynomgrade optimiert, die Geschwindigkeit ist jedoch auch in unserem Zusammenhang noch akzeptabel.

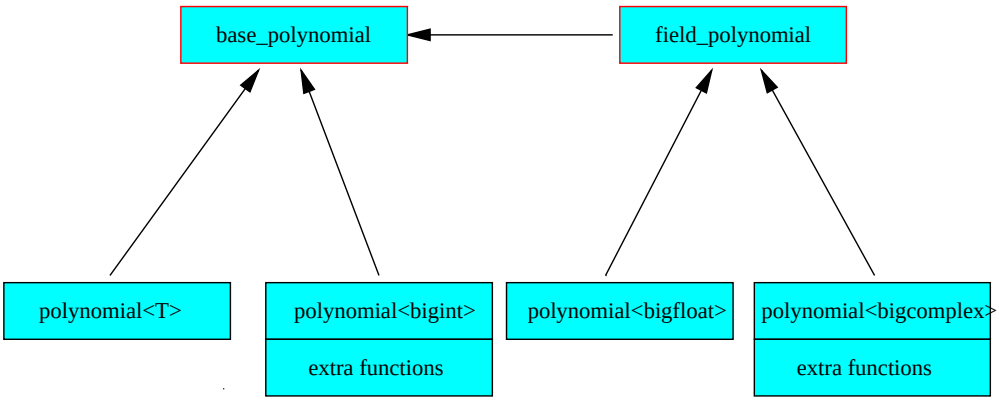

Abbildung 2.1: Die Vererbungstruktur der Polynomklassen in LiDIA.

Diese Klasse arbeitet mit modularer Arithmetik auf bigints und implementiert insbesondere auch schnelle Multiplikationsverfahren (FFT), so dass sie unabhängig neben den anderen Polynomklassen steht, jedoch dieselbe Schnittstelle anbietet.

2.3 Algebraische Zahlen

Nun wollen wir uns der Darstellung algebraischer Zahlen zuwenden. In diesem Abschnitt wollen wir zeigen, wie Elemente von Zahlkörpern und Ordnungen (im Rechner) dargestellt werden, und wollen darstellen, wie die Arithmetik implementiert ist. Dies ergibt sich dabei im wesentlichen schon aus der Darstellung der Klasse `nf_base`. Insbesondere werden wir auf die Division sowie die Berechnung von Norm und Spur eingehen.

Wir fixieren also eine $\mathbb{Q}$–Basis $\omega_1, \ldots, \omega_n$ eines algebraischer Zahlkörpers entsprechend einem Objekt aus der Klasse `nf_base`. Dann können wir eine Zahl aus diesem Zahlkörper einfach mittels seiner Koeffizienten bzgl. der Basis darstellen, d.h. eine algebraische Zahl wird durch einen Vektor $(a_1, \ldots, a_n) \in \mathbb{Q}^n$ gegeben und dieser stellt die Zahl $\alpha = \sum_{i=1}^{n} a_i \omega_i$ dar. In der Praxis ziehen wir noch einen Hauptnenner aus dem Vektor heraus, und stellen die Zahl als $\alpha = \frac{1}{d} \sum_{i=1}^{n} a'_i \omega_i$ mit $d, a'_1, \ldots, a'_n \in \mathbb{Z}$, $\quad \mathrm{ggT}(d, a'_1, \ldots, a'_n) = 1$ dar, für die Repräsentation im Rechner benutzen wir dann das Tupel $(d; a'_1, \ldots, a'_n) \in \mathbb{Z}^{n+1}$. Dies hat den Vorteil, dass wir bei allen Operationen Arithmetik über $\mathbb{Z}$ verwenden können, aber den Nachteil, dass die Zahlen größer sind, da nicht jede Komponente des Vektors einzeln gekürzt werden kann. Da jedoch über $\mathbb{Z}$ der chinesische Restsatz für kompliziertere Berechnungen angewandt werden kann,

überwiegt nach unserer Erfahrung der Vorteil.

Eine Objekt der Klasse `alg_number` in LiDIA besteht daher neben dem Verweis auf die Basis $\omega_1, \ldots, \omega_n$ in Form eines `nf_base *` aus einem Koeffizientenvektor `coeff` vom Typ `math_vector<bigint>` und einem Nenner `den` vom Typ `bigint`. Die Implementierung der Grundoperationen für algebraische Zahlen folgt der obigen Beschreibung, zur Multiplikation dient dabei entweder (2.3) oder (2.7), wobei im Falle, dass die Basis durch ein Polynom und die zugehörige Transformationsmatrix gegeben ist, zunächst die Multiplikationstafel berechnet wird. Zur Division und für einige weitere Berechnungen benutzen wir einen kleinen Trick, mit dem sich Präzisionsprobleme umgehen lassen. Diesen wollen wir nun näher betrachten. Wir benutzen die sogenannte *Darstellungsmatrix* M_α einer algebraischen Zahl α, d.h. die Matrix, die den Homomorphismus „Multiplikation mit α“ darstellt. Dabei nehmen wir ohne Einschränkung an, dass der Nenner in der Darstellung der Zahl 1 ist, so dass wir eine ganzzahlige Matrix erhalten (Ein Nenner, der verschieden von 1 ist, lässt sich leicht durch eine geeignete Vor- bzw. Nachberechnung verarbeiten: Bei der Division können wir zum Beispiel das Ergebnis durch den Nenner des Dividenden teilen und mit dem Nenner des Divisors multiplizieren).

Die Matrix M_α hat die Eigenschaft, dass das charakteristische Polynom der Matrix gleich dem charakteristischen Polynom der Zahl α im Sinne von Definition 1.1.5 ist. Damit lassen sich Norm und Spur einer algebraischen Zahl als Determinante bzw. Spur einer ganzzahligen Matrix berechnen. Die Division algebraischer Zahlen kann nun durch das Lösen eines Gleichungssystems erfolgen: Zur Berechnung von $x := \frac{\alpha}{\beta}$ ist es ausreichend, die Matrix M_β zu bestimmen und das System $M_\beta x = \alpha$ über $\mathbb{Q}$ zu lösen. In der Praxis berechnet man dabei eine Lösung von $M_\beta x = \det(M_\beta)\alpha$ über $\mathbb{Z}$ und dividiert dann durch $\det(M_\beta)$. Dabei ist zu beachten, dass es bei Verwendung des chinesischen Restsatzes ausreicht, $\det(M_\beta)$ jeweils modulo der gerade aktuellen Primzahl zu bestimmen. Dies kann während des eigentlichen Lösens passieren, so dass kein Overhead durch die Determinantenberechnung entsteht.

Damit ergibt sich die folgende Klassendefinition:

```
class alg_number{
  bigint den;
  math_vector <bigint> coeff;
  nf_base * O;
```

```
public:
  // Constructors & destructor:
  alg_number(const nf_base * O1 = nf_base::current_base);
  alg_number(const bigint &,
             const nf_base * O1 = nf_base::current_base);
  alg_number(const base_vector <bigint> &, const bigint & i,
             const nf_base * O1 = nf_base::current_base);
  alg_number(const bigint *, const bigint & i = 1,
             const nf_base * O1 = nf_base::current_base);
  alg_number(const alg_number & a);
  ~alg_number();

  alg_number & operator =(const alg_number & a);

  // member-functions
  const bigint & denominator() const;
  const math_vector <bigint> & coeff_vector() const;

  alg_number numerator() const;

  nf_base * which_base() const;
  lidia_size_t degree() const;

  bigfloat get_conjugate(lidia_size_t);
  math_vector <bigfloat> get_conjugates();

  bool is_zero() const;
  bool is_one() const;

  void normalize();
  void negate();
  void invert();
  void multiply_by_2();
  void divide_by_2();

  void assign_zero();          // Remains member of same order
```

```
void assign_one();           // Remains member of same order
void assign(const bigint &); // Remains member of same order

void assign(const alg_number &);
// Becomes member of same order as a!!

// Procedural versions of arithmetic operations:
friend void add(alg_number &,
                const alg_number &,
                const alg_number &);
friend void subtract(alg_number &,
                     const alg_number &,
                     const alg_number &);
friend void multiply(alg_number &,
                     const alg_number &,
                     const alg_number &);
friend void divide(alg_number &,
                   const alg_number &,
                   const alg_number &);

friend void add(alg_number &,
                const alg_number &,
                const bigint &);
friend void subtract(alg_number &,
                     const alg_number &,
                     const bigint &);
friend void multiply(alg_number &,
                     const alg_number &,
                     const bigint &);
friend void divide(alg_number &,
                   const alg_number &,
                   const bigint &);

friend void add(alg_number &,
                const bigint &,
                const alg_number &);
```

```
friend void subtract(alg_number &,
                     const bigint &,
                     const alg_number &);
friend void multiply(alg_number &,
                     const bigint &,
                     const alg_number &);
friend void divide(alg_number &,
                   const bigint &,
                   const alg_number &);

friend void power(alg_number &,
                  const alg_number &,
                  const bigint &);
friend void power_mod_p(alg_number &,
                        const alg_number &,
                        const bigint &,
                        const bigint &);

// arithmetic operators:
friend alg_number
operator -(const alg_number &);
friend alg_number
operator +(const alg_number &, const alg_number &);
friend alg_number
operator +(const alg_number &, const bigint &);
friend alg_number
operator +(const bigint &, const alg_number &);
friend alg_number
operator -(const alg_number &, const alg_number &);
friend alg_number
operator -(const alg_number &, const bigint &);
friend alg_number
operator -(const bigint &, const alg_number &);
friend alg_number
operator *(const alg_number &, const alg_number &);
friend alg_number
```

```
operator *(const alg_number &, const bigint &);
friend alg_number
operator *(const bigint &, const alg_number &);
friend alg_number
operator /(const alg_number &, const alg_number &);
friend alg_number
operator /(const alg_number &, const bigint &);
friend alg_number
operator /(const bigint &, const alg_number &);

alg_number & operator +=(const alg_number & a);
alg_number & operator +=(const bigint & a);
alg_number & operator -=(const alg_number & a);
alg_number & operator -=(const bigint & a);
alg_number & operator *=(const alg_number & a);
alg_number & operator *=(const bigint & a);
alg_number & operator /=(const alg_number & a);
alg_number & operator /=(const bigint & a);

// Comparisions:
// By now, only comparision of numbers
// over the same order is implemented.
friend bool operator ==(const alg_number&, const alg_number&);
friend bool operator !=(const alg_number&, const alg_number&);

bool operator ! () const
{return is_zero();}

// Some number-theoretic functions:
friend lidia_size_t degree(const alg_number &);
friend bigint_matrix rep_matrix(const alg_number &);
friend bigrational norm(const alg_number &);  // Norm
friend bigrational trace(const alg_number &); // Trace
friend polynomial <bigint> charpoly(const alg_number &);
                  // characteristic polynomial
```

```
  // other functions:
  friend void negate(alg_number &, const alg_number &);
  friend void invert(alg_number &, const alg_number &);
  friend alg_number inverse(const alg_number &);
  friend const bigint & denominator(const alg_number &);
  friend const math_vector <bigint> &
  coeff_vector(const alg_number &);
  friend alg_number numerator(const alg_number &);
  friend nf_base * which_base(const alg_number &);
  friend void square(alg_number &, const alg_number &);
  friend void swap(alg_number &, alg_number &);

  // random numbers
  void randomize(const bigint &);

  // In-/Output:
  friend ostream& operator<<(ostream&, const alg_number&);
  friend istream& operator>>(istream&, alg_number&);

  // friends:
  friend class module;
  friend void multiply(alg_ideal &, const alg_ideal &,
                       const alg_number &);
  friend void multiply(alg_ideal &, const alg_number &,
                       const alg_ideal &);
  friend void divide(alg_ideal &, const alg_ideal &,
                     const alg_number &);

};
```

Die Namen fast aller Funktionen sind selbsterklärend und implementieren die bisher beschriebenen Algorithmen. Einer weiteren Erläuterung bedürfen wohl nur die Funktionen `get_conjugates()` und `get_conjugate(i)`. Diese geben den Konjugiertenvektor der algebraischen Zahl bzw. die i-te Komponente dieses Vektors zurück. Dabei wird der Konjugiertenvektor jeweils durch Multiplikation der Matrix aus den Konjugiertenvektoren der Basis mit dem Koeffizientenvektor errechnet. Sind die Konjugiertenvektoren der Basis noch nicht

bekannt, so werden sie an dieser Stelle berechnet und (wenn möglich, d.h. wenn der Compiler das Schlüsselwort `mutable` unterstützt) abgespeichert.

2.4 Zahlkörper und Ordnungen

Nun wenden wir uns der Frage zu, wie wir Zahlkörper und Ordnungen in LiDIA behandeln. Dazu erinnern wir daran, dass ein Zahlkörper stets ein $\mathbb{Q}$–Vektorraum der Dimension des Körpergrads ist bzw. dass eine Ordnung ein freier $\mathbb{Z}$–Modul ist, dessen Rang gerade dem Körpergrad entspricht. Daher fixiert man in der Mathematik stets eine $\mathbb{Z}$–Basis dieses Vektorraums bzw. Moduls (im Falle von $\mathbb{Q}(\rho)$ ist dies meist $\{1 = \rho^0, \rho^1, \ldots, \rho^{n-1}\}$) und stellt algebraische Zahlen als Linearkombinationen der Basiselemente dar.

Um die Gemeinsamkeiten zwischen Zahlkörper und Ordnung stärker zu betonen, verlangen wir, wie schon gesehen, dass auch die Basis eines Zahlkörpers stets in Form von algebraisch ganzen Zahlen gegeben ist, so dass wir bei allen Darstellungsarten ganze statt rationaler Zahlen benutzen können.

Aus dieser zusätzlichen Bedingung ergibt sich insbesondere, dass wir für das Polynom aus Definition 1.1.1 die zusätzliche Anforderung stellen, dass es sich um ein normiertes ganzzahliges Polynom handeln muss, was o.B.d.A. möglich ist: Ist ein Körper $K = \mathbb{Q}(\rho)$ gegeben, wobei $\rho = \frac{\rho'}{n}$ für eine algebraisch ganze Zahl ρ' und eine natürliche Zahl n gilt, so ist $K = \mathbb{Q}(\rho')$ und das Minimalpolynom von ρ' erfüllt unsere Bedingungen.

Eine Basis aus algebraisch ganzen Zahlen, bzw. die Information darüber, wie man Basiselemente multipliziert, haben wir bereits in der Klasse `nf_base` gekapselt. Diese Konzeption hat zudem den Vorteil, dass wir auf diese Weise die Lebensdauer eines Objekts der Klasse `nf_base` dynamisch handhaben können, so dass z.B. Zahlen oder Ideale aus einer Ordnung selbst dann noch miteinander multipliziert werden können, wenn die Ordnung selbst nicht mehr im Speicher liegt. Dazu erhalten alle Strukturen, die die in `nf_base` enthaltene Information benötigen, einen Zeiger auf ein solches Objekt und in der Klasse `nf_base` selbst werden wir mittels Reference–Counting darüber Buch führen, wieviele Objekte die dort gespeicherte Information noch benötigen könnten und erst im Falle, dass diese Zahl auf 0 sinkt, das Objekt löschen. Dabei erweist es sich dank des Klassenkonzeptes als ausreichend, das Zählen der Referenzen in Konstruktoren und Destruktoren zu implementieren.

Die Klassen `number_field` und `order`

Zahlkörper und Ordnungen werden nun im wesentlichen durch einen Pointer auf ein Objekt der Klasse `nf_base` realisiert. Die Verwendung eines Pointers ist dabei wesentlich, da hierdurch eine Entkoppelung der Lebensdauer von Objekten der Klasse `nf_base` und den Objekten der Klassen `number_field` und `order` erreicht wird, so dass ein Objekt vom Typ `nf_base` gleichzeitig als Basis eines Zahlkörpers und einer Ordnung dienen kann. Darüber hinaus können auch algebraische Zahlen und Moduln bzw. Ideale sinnvoll definiert werden, da man sich aufgrund des des *Reference–Countings* keine Sorgen über die Lebensdauer eines Objekts der Klasse `nf_base` machen muss. Für die Implementierung ist es dann ausreichend, ein Objekt der Klasse `nf_base` anzugeben, um eine algebraische Zahl bzw. einen Modul bzw. ein Ideal zu definieren. In der Sprache der Mathematik gibt man hier aber üblicherweise den Zahlkörper oder die Ordnung an, in dem eine Zahl bzw. ein Modul oder ein Ideal liegen soll. Um diese Notation zu unterstützen, wird in einem solchen Fall durch einen impliziten Cast–Operator automatisch der Verweis auf das Objekt der Klasse `nf_base` aus dem angegebenen Objekt der Klasse `number_field` oder `order` extrahiert.

Darüber hinaus hat es sich auch eingebürgert, vom Zahlkörper zu sprechen, wenn eigentlich die Maximalordnung des Zahlkörpers gemeint ist. So spricht man z.B. von der Diskriminante eines Zahlkörpers bzw. von der Klassenzahl eines Zahlkörpers. Auch diese Sprechweise wird von unseren Klassen unterstützt, indem in solchen Fällen zunächst implizit die Maximalordnung berechnet wird und dann die eigentliche Berechnung auf der Maximalordnung ausgeführt wird.

Damit ergeben sich die folgenden Klassendefinitionen

```
class number_field{
  nf_base * base;        // pointer to the base of O

public:
  // Contructors & destructor:
  number_field();
  number_field(const polynomial < bigint > & p);
  number_field(const bigint *v, lidia_size_t deg);
  number_field(const order &);
  number_field(const number_field &);
  ~number_field();
```

```
// Cast operator
operator nf_base *() const
{ return base;}

// member functions:
const bigfloat & get_conjugate(lidia_size_t, lidia_size_t);
const bigfloat_matrix & get_conjugates();

const polynomial <bigint> & which_polynomial() const;

nf_base * which_base() const;

lidia_size_t degree() const;

lidia_size_t no_of_real_embeddings() const;

void assign(const number_field & K);

number_field & operator =(const number_field & F);

// Comparisions:
// Since deciding whether fields are isomorphic is
// difficult, we consider fields to be equal only,
// if they are generated by the same polynomial.
friend bool operator ==(const number_field &,
                        const number_field &);
friend bool operator !=(const number_field &,
                        const number_field &);
friend bool operator <=(const number_field &,
                        const number_field &);
friend bool operator <(const number_field &,
                       const number_field &);
friend bool operator >=(const number_field &,
                        const number_field &);
friend bool operator >(const number_field &,
```

```
                        const number_field &);

  // friend functions:
  friend const polynomial < bigint > &
  which_polynomial(const number_field &);
  friend nf_base * which_base(const number_field &);
  friend lidia_size_t degree(const number_field &);
  friend lidia_size_t
  no_of_real_embeddings(const number_field &);
  // In-/Output:
  friend ostream&
  operator<<(ostream&, const number_field&); // jeweils Polynom
  friend istream&
  operator>>(istream&, number_field&);       // ein-, ausgeben
};



class order{
  // Components of the data - type:
  mutable bigint discriminant;
  nf_base * base;        // pointer to the base of O

  // Internal routine for the work involved in comparisons
  void compare(const order&, bool &, bool &) const;

public:
  // Contructors & destructor:
  order(const nf_base * base1 = nf_base::current_base);
  order(const polynomial<bigint> & p,
        const base_matrix <bigint> & = bigint_matrix(),
        const bigint & = 1);
  // initialize with equation order (default)
  // or with transformed eq. order.
  order(const base_matrix <bigint> &);  // table uebergeben
  order(const base_matrix <bigint> &, const bigint &,
        const nf_base * base1 = nf_base::current_base);
```

```
// initialize with base1 transformed by a trafo.
order(const order &);                  // copy constructor
order(const number_field &);           // Maximal order !
~order();

order & operator =(const order & O);

// member functions:
     // Accessing Information:

const bigint_matrix & base_numerator() const;

const bigint & base_denominator() const;

const polynomial <bigint> & which_polynomial() const;

nf_base * which_base() const

const bigint & MT(lidia_size_t,lidia_size_t,lidia_size_t);
// accessing mltiplication table

const bigfloat & get_conjugate(lidia_size_t, lidia_size_t);
const bigfloat_matrix & get_conjugates();

lidia_size_t degree() const;
// degree of extension

lidia_size_t no_of_real_embeddings() const;

void assign(const order &);

// Cast operator:
operator alg_ideal() const;

operator nf_base *() const;
```

```
// Comparisions:
// We are interested in comparing orders only if they are
// over the same field! So nothing else is supported!!
friend bool operator ==(const order&, const order&);
friend bool operator !=(const order&, const order&);
friend bool operator <=(const order&, const order&);
friend bool operator <(const order&, const order&);
friend bool operator >=(const order&, const order&);
friend bool operator >(const order&, const order&);

// friend functions:
friend const bigint_matrix &
base_numerator(const order &);
friend const bigint & base_denominator(const order &);
friend const polynomial <bigint> &
which_polynomial(const order &);
friend nf_base * which_base(const order &);
friend void swap(order &, order &);

// Number-theoretic functions:
friend lidia_size_t
degree(const order & O);  // degree of field extension
friend lidia_size_t
no_of_real_embeddings(const order & O);
friend const bigint & disc(const order & O); // discriminant
module pseudo_radical(const bigint & p) const;
                                  // computes pseudo-radical
bool dedekind(const bigint & p,
              polynomial < bigint > & h2) const;
  // uses Dedekind criterion for prime p; returns true,
  // if p is an index divisor and returns a generator of
  // the p-radical if this is the case.

order maximize(const bigint & p) const;
            // maximizes order at p.
order maximize() const;     // full maximization (round2)
```

```
  order maximize2() const;     // full maximization (round2)
                               // (sligtly different strategy).

  // In-/Output:
  friend ostream& operator<<(ostream &, const order &);
  friend istream& operator>>(istream &, order &);
};
```

Neben den evidenten Zugriffsfunktionen ist insbesondere der Konstruktor, der aus einem Zahlkörper eine Ordnung - nämlich seine Maximalordnung - konstruiert von Interesse. Dieser ermöglicht es, überall da, wo man einer Funktion eigentlich einen Ganzheitsring als Argument übergeben müsste, genauso gut den Zahlkörper einzusetzen, da dann eine automatische Konvertierung stattfindet. Damit ergibt sich in natürlicher Weise die Berechnung der Körperdiskriminanten oder der Klassenzahl eines Zahlkörpers.

Die Funktionen `get_conjugates()` bzw. `get_conjugate(i,j)` geben eine Matrix, deren i-te Spalte der Konjugiertenvektor des i-ten Basiselements ist, bzw. einen Eintrag aus dieser Matrix zurück. Im Bedarfsfall wird dabei zunächst mittels der oben beschriebenen Funktion `compute_conjugates()` aus der Klasse `nf_base` die Berechnung dieser Werte durchgeführt.

Besondere Beachtung verdienen natürlich auch die Funktionen, die zur Maximalordnungsberechnung genutzt werden. Die Funktionen `maximize()` und `maximize2()` berechnen zu einer gegebenen Ordnung die Maximalordnung. Diesen Algorithmen wollen wir den nächsten Abschnitt widmen.

2.5 Berechnung der Maximalordnung

Die naive Annahme, die Berechnung der Maximalordnung sei fundamentaler als die Idealarithmetik, erweist sich als falsch, obwohl man erwarten könnte, dass man zunächst den Ring bestimmen sollte, in dem die weiteren Berechnungen stattfinden. Tatsächlich aber werden wir Idealarithmetik benötigen, um die Maximalordnung zu bestimmen. Die benötigten Algorithmen der Idealarithmetik, der wir uns im nächsten Kapitel zuwenden wollen, wollen wir an dieser Stelle als „Black Box“ voraussetzen.

Der Algorithmus zur Maximalordnungsberechnung basiert darauf, dass der Multiplikatorenring eines nicht invertierbaren Ideals eine Ordnung ist, die die Ordnung, in der das Ideal liegt, echt umfasst. Die Grundidee des Algorithmus

besteht nun darin, sukzessive geeignete nicht invertierbare Ideale zu bestimmen und ihre Multiplikatorenringe zu berechnen. Dazu benötigen wir zunächst noch folgenden Hilfsbegriff:

2.5.1. Definition (p–Radikal)
Sei $p \in \mathbb{N}$. Das Nilradikal von $\mathcal{O}/p\mathcal{O}$, d.h. das Ideal aller nilpotenten Elemente in $\mathcal{O}/p\mathcal{O}$, bezeichnen wir als p–Radikal *I_p von $\mathcal{O}$.*

Laut [BLJ], Proposition 5.1 gilt nun

2.5.2. Satz
Sei $\mathcal{O}$ eine Ordnung in einem algebraischen Zahlkörper K. Für den Multiplikatorenring $\mathcal{O}'$ des p–Radikals gilt: $\mathcal{O}'$ ist eine Ordnung von K, die $\mathcal{O}$ enthält mit $[\mathcal{O}' : \mathcal{O}] \mid p^{n-1}$. Falls insbesondere $\mathcal{O}' = \mathcal{O}$ gilt, so ist $\mathrm{ggT}([\mathcal{O}_K : \mathcal{O}], l) = 1$ für alle Primzahlen l mit $l \mid p, l^2 \nmid p$.

Damit könnte ein erster Entwurf des Algorithmus zur Berechnung der Maximalordnung wie folgt aussehen:

2.5.3. Algorithmus

Berechnung der Maximalordnung EINGABE: Eine Ordnung $\mathcal{O}$ eines Zahlkörpers K AUSGABE: Die Maximalordnung $\mathcal{O}_K$
(1) Berechne eine Faktorisierung $\Delta_{\mathcal{O}} = p_1^{e_1} \cdots p_n^{e_n}$, $p_1, \ldots, p_n$ prim

(2) $p = \prod_{\substack{i=1 \\ e_i>1}}^{n} p_i$
(3) **repeat**
(4) $\mathcal{O}' = \mathcal{O}$
(5) Berechne das p–Radikal I_p
(6) Berechne mit Algorithmus 3.1.10 den Multiplikatorenring $\mathcal{O}$ von I_p
(7) **until** $(\mathcal{O}' \neq \mathcal{O})$
(8) return $\mathcal{O}$

Man beachte allerdings, das wir noch nicht beschrieben haben, wie man das p–Radikal berechnen kann. Tatsächlich scheitert hieran sogar der soeben beschriebene Entwurf. Zudem gibt es mit dem sogenannten Dedekind–Test speziell für Primzahlen p auch eine wesentlich effizientere Methode, um zu testen, ob $\mathrm{ggT}([\mathcal{O}_K : \mathcal{O}], p) = 1$ gilt, bzw. falls nicht, um einen ersten Maximierungsschritt auszuführen. Dieser Test beruht auf dem folgenden Satz:

2.5.4. Satz

Sei $\mathcal{O}$ eine Ordnung eines algebraischen Zahlkörpers K, $A \in \mathbb{Z}[X]$ das normierte Minimalpolynom einer algebraisch ganzen Zahl ρ mit $K = \mathbb{Q}(\rho)$ und p eine Primzahl und es gelte $\mathrm{ggT}([\mathcal{O} : \mathbb{Z}[\rho]], p) = 1$. Es bezeichne $\bar{\cdot}$ die Reduktion modulo p, und es sei

$$\overline{A}(X) = \prod_{i=1}^{k} \overline{t_i}(X)^{e_i}$$

eine Zerlegung von $\overline{A}(X)$ in irreduzible Faktoren in $\mathbb{F}_p[X]$. Es bezeichne

$$g(X) = \prod_{i=1}^{k} t_i(X),$$

wobei $t_i \in \mathbb{Z}[X]$ ein beliebiges normiertes Urbild von $\overline{t_i}$ (unter $\bar{\cdot}$)ist. Dann gilt

1. *Für das p–Radikal gilt:*
$$I_p = p\mathcal{O} + g(\rho)\mathcal{O}.$$
2. *Sei $h \in \mathbb{Z}[X]$ ein beliebiges normiertes Urbild von $\overline{A}(X)/\overline{g}(X)$ (unter $\bar{\cdot}$). Dann gilt mit $f(X) = (g(X)h(X) - A(X))/p \in \mathbb{Z}[X]$:*
$$\mathrm{ggT}([\mathcal{O}_K : \mathcal{O}], p) = 1 \Leftrightarrow \mathrm{ggT}(\overline{f}, \overline{g}, \overline{h}) = 1 \text{ in } \mathbb{F}_p[X].$$

3. *Sei* $U \in \mathbb{Z}[X]$ *ein beliebiges normiertes Urbild von* $\overline{A}(X)/\operatorname{ggT}(\overline{f}, \overline{g}, \overline{h})$ *(unter* $\bar{\cdot}$*). Dann gilt für den Multiplikatorenring* $\mathcal{O}'$ *von* I_p*:*

$$\mathcal{O}' = \mathcal{O} + \frac{1}{p}U(\rho)\mathcal{O}$$

und $[\mathcal{O}' : \mathcal{O}] = p^m$ *mit* $m = \deg(\operatorname{ggT}(\overline{f}, \overline{g}, \overline{h}))$.

Zum Beweis vergleiche [Coh95], Kapitel 6.1.2.

Damit ist für den Spezialfall, dass die Ordnung noch nicht für p maximiert wurde und p eine Primzahl ist, klar, wie der erste Maximierungsschritt ausgeführt werden kann. Wir haben sogar schon eine explizite Formel für den Multiplikatorenring. Für den allgemeinen Fall fehlt uns noch ein Algorithmus zur Berechnung des p–Radikals. Diesen liefert uns der folgende Satz, der dieses Problem auf ein Problem der linearen Algebra über $\mathbb{Z}/p\mathbb{Z}$ zurückführt, das wir mit den Mitteln des nächsten Kapitels lösen werden.

2.5.5. Satz
Sei $\mathcal{O}$ *eine Ordnung eines algebraischen Zahlkörpers* K *mit* $[K : \mathbb{Q}] = n$*, sei* p *eine natürliche Zahl und bezeichne* $\bar{\cdot}$ *wieder Reduktion modulo* p*. Dann gilt für das* p*–Radikal* I_p*:*

1. *Falls* p *eine Primzahl ist und* j *so gewählt wurde, dass gilt* $p^j \geq n$*, so ist* I_p *der Kern des Homomorphismus*

$$\mathcal{O} \longrightarrow \mathcal{O}/p\mathcal{O}, \alpha \longmapsto \overline{\alpha}^{q^j}.$$

2. *Falls* $\operatorname{ggT}(n!, p) = 1$ *ist, so ist* I_p *der Kern des Homomorphimus*

$$\mathcal{O} \longrightarrow Hom(\mathcal{O}/p\mathcal{O}, \mathbb{Z}/p\mathbb{Z}), \alpha \longmapsto (\overline{\beta} \mapsto Tr(\overline{\alpha\beta})).$$

Der Beweis ergibt sich aus [BLJ], Proposition 3.7 und Proposition 3.9. Man beachte insbesondere, dass für Primzahlen größer n beide Verfahren tatsächlich das gleiche Ergebnis finden, dass wir aber für zusammengesetzte Zahlen, die einen Primfaktor $\leq n$ enthalten, gar kein Berechnungsverfahren haben.

Abschließend können wir nun den verfeinerten Algorithmus zur Berechnung der Maximalordnung formulieren. Dabei sei `dedekind(`$\mathcal{O}$`, `$\mathcal{O}$`', p)` eine Funktion, die `false` zurückgibt, falls der Dedekind–Test zeigt, dass bereits $\operatorname{ggT}([\mathcal{O}_K : \mathcal{O}], p) = 1$ gilt und `true` sonst. Dabei sei im zweiten Falle $\mathcal{O}$' bereits das Resultat des ersten Maximierungsschritts.

2.5.6. Algorithmus

Berechnung der Maximalordnung

EINGABE: Eine Ordnung $\mathcal{O}$ eines Zahlkörpers K vom Grad n
AUSGABE: Die Maximalordnung $\mathcal{O}_K$

(1) Berechne eine Faktorisierung $\Delta_{\mathcal{O}} = p_1^{e_1} \cdots p_n^{e_n}$, $p_1, \ldots, p_n$ prim
(2) $p = 1$
(3) **for** $(i = 1; i <= n; i++)$ **do**
(4) **if** (dedekind$(\mathcal{O}, \mathcal{O}', p)$) **then**
(5) **if** $([\mathcal{O}' : \mathcal{O}]^2 < p_i^{e_i - 1})$ **then**
(6) **if** $(p_i > n)$ **then**
(7) $p = p \cdot p_i$
(8) **else**
(9) **repeat**
(10) $\mathcal{O} = \mathcal{O}'$
(11) Berechne das p_i–Radikal I_{p_i} von $\mathcal{O}$
(12) Berechne mit Algorithmus 3.1.10 den Multiplikatorenring $\mathcal{O}'$ von I_{p_i}
(13) **until** $(\mathcal{O}' == \mathcal{O})$
(14) **fi**
(15) **fi**
(16) $\mathcal{O} = \mathcal{O}'$
(17) **fi**
(18) **od**
(19) **if** $(p > 1)$ **then**
(20) **repeat**
(21) $\mathcal{O} = \mathcal{O}'$
(22) Berechne das p–Radikal I_p von $\mathcal{O}$
(23) Berechne mit Algorithmus 3.1.10 den Multiplikatorenring $\mathcal{O}'$ von I_p
(24) **until** $(\mathcal{O}' == \mathcal{O})$

(25) **fi**
(26) return $\mathcal{O}$

Dabei geht die Funktion `maximize()` so vor, wie in Algorithmus 2.5.6 beschrieben, wobei die Diskriminante mittels der Funktion `disc` ausgerechnet wird, `dedekind` den Dedekind–Test implementiert und von `maximize(p)` die Maximierung bzgl. einer quadratfreien Zahl p vorgenommen wird, die entweder prim ist oder nur Faktoren enthält, die größer als der Körpergrad sind. Die Funktion `pseudo_radical` übernimmt dabei die Radikalberechnung, die Berechnung des Multiplikatorenrings ist in der Klasse zur Darstellung von Moduln realisiert. Der Dedekind–Test findet allerdings nur statt, falls in dem Objekt der Klasse `nf_base`, das zur Darstellung der Ordnung benutzt wird, ein Körperpolynom eingetragen ist. Im Unterschied zu `maximize()` benutzt `maximize2()` eine etwas andere Strategie zur Maximierung. Statt zuerst alle Primzahlen, die den Körpergrad übersteigen, zusammen zu multiplizieren, um dann in einem Maximierungsschritt alle Arbeit zu erledigen, wird hier im Anschluß an den eventuellen Dedekind–Test direkt für jede Primzahl p die Maximierung mittels `maximize(p)` vorgenommen. Dies ist der ursprüngliche sogenannte *Round–2–Algorithmus* von Zassenhaus (vgl. [Coh95], Kapitel 6.1.3 und 6.1.4). Daneben gibt es noch den *Round–4–Algorithmus*. Dieser ist in Spezialfällen deutlich effizienter, insbesondere, wenn die Diskriminante eine hohe Potenz einer Primzahl enthält. Dies ist jedoch nur sehr selten der Fall, wenn man es nicht gezielt darauf anlegt, solche Beispiele zu konstruieren. Tatsächlich ist im allgemeinen die Faktorisierung der Diskriminante die bei weitem zeitaufwändigste Teilaufgabe, wie man in Tabelle 2.1 sieht. Wenn man nicht gerade Körper mit sehr kleiner Diskriminante behandelt, benötigt die linearen Algebra, die man im Round–2–Algorithmus ausführen muss, ohnehin nur einen Bruchteil, im Extremfall sogar nur wenige Prozent der Laufzeit für die Faktorisierung. Daher lohnt sich unserer Auffassung nach der deutliche höhere Implementierungsaufwand für den Round–4–Algorithmus nicht. Um dies zu illustrieren, haben wir an einigen Beispielen die Zeit zur Berechnung der Diskriminanten, zur Faktorisierung derselben und für die eigentliche Maximierung der Ordnung mittels unseres modifizierten Round–2–Algorithmus tabelliert.

Insbesondere ist bei diesen Zeiten allerdings zu beachten, dass die Zeiten zur Faktorisierung starken Schwankungen unterliegen. Häufig können diese auch bei

f	T_d	T_f	T_m
$x^3 - 4414720139365147385322203250687$	0.2s	0.48s	0.6s
$x^3 - 44147201418278263343553971153985$	0.0s	0.88s	0.3s
$x^3 - 441472014238575797358264732597668$	0.0s	1.5s	0.5s
$x^3 - 441472014238575797358264732597670$	0.0s	0.49s	0.2s
$x^3 - 44147201427559772304963099328971 29$	0.1s	1.54s	0.3s
$x^3 - 441472014295202904501258370560230 01$	0.1s	1.50s	0.3s
$x^3 - 441472014312906430669804999858773 7\backslash$ $6848657163235155930219823088236 7$	0.1s	1m 2s	0.3s
$x^3 - 139605708845123841207629852606935 1\backslash$ $4275197878812278037384709162612986$	0.1s	397m	0.2s
$x^4 - 120440771446949753882585617$	0.1s	2.88s	0.9s
$x^4 - 209700041469577058114695697$	0.0s	0.93s	0.9s
$x^4 - 365109704143734045008630017$	0.1s	3.38s	0.9s
$x^4 - 635695964911209990157111297$	0.1s	3.13s	0.9s
$x^4 - 110681517674872082679466881 7$	0.1s	1.6s	0.9s
$x^4 - 192708491863759912960000000 1$	0.1s	3.75s	0.9s
$x^6 - 1566742110959875129345$	0.2s	0.93s	0.30s
$x^6 - 2917096519063103999999$	0.3s	0.25s	0.15s
$x^6 - 2917096519063104000001$	0.2s	4.14s	0.30s
$x^6 - 5395603628295198019583$	0.3s	0.21s	0.15s
$x^6 - 539560362829519801958 5$	0.3s	0.66s	0.31s
$x^6 - 284560482437949172305084825 7$	0.3s	3.32s	0.33s
$x^6 - 516781442640795158534399999 9$	0.2s	0.75s	0.15s
$x^6 - 516781442640795158534400000 1$	0.3s	5.12s	0.34s
$x^6 - 938812087213265364444899999 9$	0.2s	6.11s	0.14s
$x^6 - 938812087213265364444900000 1$	0.2s	4.83s	0.33s
$x^6 - 170084380525520362709872803 83$	0.2s	0.78s	0.15s
$x^6 - 247820883089418472856326273626681 5$	0.2s	0.84s	0.15s
$x^6 - 247820883089418472856326273626681 7$	0.2s	5.49s	0.35s
$x^6 - 448921154482111771036650985200134 3$	0.2s	5.25s	0.15s
$x^6 - 448921154482111771036650985200134 5$	0.3s	1.41s	0.35s
$x^6 - 813253729114622644223492505599999 9$	0.3s	0.76s	0.15s
$x^6 - 813253729114622644223492505600000 1$	0.3s	4.54s	0.34s

Tabelle 2.1: Laufzeiten zur Maximalordnungsberechnung

T_d: Zeit zur Berechnung der Diskriminante, T_f: Zeit zur Faktorisierung, T_m: Zeit zur Maximierung

ein und demselben Körper auftreten, je nachdem, wieviel Glück oder Pech man bei den randomisierten Algorithmen zur Faktorisierung hat.

Kapitel 3

Idealarithmetik in Zahlkörpern und Lineare Algebra in $\mathbb{Z}/m\mathbb{Z}$

In diesem Kapitel gehen wir auf die Klassen `module` und `alg_ideal` unserer Implementierung ein. Dabei zeigen wir die Reduktion der Arithmetik auf Probleme der Linearen Algebra, die von uns implementiert wurde. Anschließend wollen wir zeigen, wie man diese Probleme der Linearen Algebra lösen kann und wie wir diese Algorithmen implementiert haben. Dabei werden wir den allgemeinen Fall beschreiben, der unserer Implementierung zu Grunde liegt.

3.1 Idealarithmetik

Zunächst wollen wir zeigen, wie man die grundlegenden Rechenoperationen für Moduln bzw. Ideale realisieren kann. Wo immer dies ohne einen Verlust an Effizienz möglich ist, wollen wir dabei auf den allgemeineren Fall der $\mathbb{Z}$–Moduln eingehen, wo es für Ideale effizientere Methoden als für Moduln gibt, werden wir nur den spezielleren Fall betrachten.

Zunächst wollen wir allerdings eine Vereinfachung vornehmen: Wir werden stets annehmen, dass die Moduln oder Ideale, die wir betrachten, Teilmengen der Ordnung sind, d.h. wir setzen voraus, dass die Nenner stets 1 sind. Dies ist zulässig, da die Nenner mittels geeigneter Vorberechnungen berücksichtigt werden können. Sind etwa $\mathfrak{a} = \mathfrak{a}'/a$ und $\mathfrak{b} = \mathfrak{b}'/b$ zwei Moduln, so dass $\mathfrak{a}'$ und

$\mathfrak{b}'$ in $\mathcal{O}$ liegen so gilt:

$$\begin{aligned}
\mathfrak{a}+\mathfrak{b} &= \left(\frac{b}{\mathrm{ggT}(a,b)}\mathfrak{a}' + \frac{a}{\mathrm{ggT}(a,b)}\mathfrak{b}'\right)\cdot\frac{1}{\mathrm{kgV}(a,b)}\\
\mathfrak{a}\cap\mathfrak{b} &= \left(\frac{b}{\mathrm{ggT}(a,b)}\mathfrak{a}' \cap \frac{a}{\mathrm{ggT}(a,b)}\mathfrak{b}'\right)\cdot\frac{1}{\mathrm{kgV}(a,b)}\\
\mathfrak{a}\cdot\mathfrak{b} &= \left(\mathfrak{a}'\cdot\mathfrak{b}'\right)\frac{1}{ab}\\
(\mathfrak{a}:\mathfrak{b}) &= (\mathfrak{a}':\mathfrak{b}')\frac{b}{a}
\end{aligned}$$

3.1.1 Darstellung von Moduln und Idealen

Von großer Bedeutung für die Algorithmen ist die Art und Weise, in der Ideale dargestellt werden. Hierfür sind in der Literatur verschiedene Methoden bekannt, die auch verschiedene Implementierungen erfordern. Dabei erweist sich insbesondere die Idealdivision in allen Darstellungen als Problem.

In dieser Arbeit schlagen wir eine neue Darstellung vorschlagen, die es erlaubt, in einfacher Weise die später beschriebenen Algorithmen für die Lineare Algebra über $\mathbb{Z}/m\mathbb{Z}$ zu benutzen, um die Idealarithmetik zu realisieren. In naiver Weise können wir einen Modul bzw. ein Ideal durch eine $\mathbb{Z}$–Basis aus algebraischen Zahlen beschreiben. Diese Zahlen werden typischerweise selbst wieder als Koeffizientenvektor bzgl. einer Körperbasis beschrieben. Ordnen wir diese Vektoren als Spalten einer Matrix an, so zeigt sich, dass diese Matrix gerade die Transformationsmatrix aus $\mathbb{Q}^{n\times n}$ ist, durch die die Basis des Ideals aus der Basis der Ordnung hervorgeht. Diese Matrix ist jedoch wie jede Basis eines Moduls nur bis auf unimodulare Transformationen eindeutig bestimmt. Um Eindeutigkeit zu erreichen, kann man etwa verlangen, dass stets ein Hauptnenner a vor die Matrix gezogen wird und die verbleibende Matrix aus $\mathbb{Z}^{n\times n}$ in Hermite–Normalform gegeben wird.

Damit hat man den Nachteil, dass man bei Körpergrad n nun $\frac{n(n+1)}{2}$ Zahlen zur Beschreibung eines Ideals benötigt, wobei diese Zahlen offensichtlich durch die Norm des Ideals beschränkt sind. Der Vorteil dieser Darstellungsweise liegt darin, dass die Probleme der Idealarithmetik nun auf Kern– und Bildberechnungen über $\mathbb{Z}$ reduziert werden können, wie dies etwa in [Coh95] geschieht.

Wir haben eine Alternative entwickelt, mit der wir einerseits die Darstellung verkürzen und andererseits die Kern– und Bildberechnungen über $\mathbb{Z}$ durch die effizienteren Kern– und Bildberechnungen über $\mathbb{Z}/m\mathbb{Z}$ ersetzen können.

Um unsere Darstellung erklären zu können, benötigen wir zunächst den Begriff des Exponenten.

3.1.1. Definition
Sei $M \subseteq \mathcal{O}$ ein vollständiger Modul in einem algebraischen Zahlkörper. Die kleinste positive ganze Zahl d mit $d \cdot \mathcal{O} \subseteq M$ heißt Exponent des Moduls M, in Zeichen $\exp(M) = d$.

Für Moduln $N \subseteq \mathcal{O}$, die nicht vollständig sind, setzen wir $\exp(N) := 0$.

Für $d \in \mathbb{N}$ gibt es eine Bijektion zwischen Moduln $M \subseteq \mathcal{O}$ mit $\exp(M)|d$ und den Moduln in $\mathcal{O}/d\mathcal{O}$. (Ein gegebener Modul $M/d\mathcal{O}$ mit Basis $(\bar{a}_1, \ldots, \bar{a}_m)$ entspricht dem folgenden Modul M: Lifte $\bar{a}_i$ zu einer Zahl $a_i \in \mathcal{O}$ $(1 \leq i \leq m)$ und setze $M = a_1\mathbb{Z} + \cdots a_m\mathbb{Z} + d\mathcal{O}$.)

3.1.2. Beispiel
Seien $\omega_1, \ldots, \omega_4$ wie in Beispiel 2.1.2. Dann ist die Menge $\mathfrak{a} = 21\mathbb{Z} + 21\omega_2\mathbb{Z} + (17 + 3\omega_2 + \omega_3)\mathbb{Z} + (20 + 4\omega_2 + \omega_4)\mathbb{Z}$ ein Ideal in der Maximalordnung O_K von $K = \mathbb{Q}(\sqrt{-2} + \sqrt{5})$ und es ist $\exp(\mathfrak{a}) = 21$. Daher können wir dieses Ideal einfacher auch als $(17 + 3\omega_2 + \omega_3)\mathbb{Z} + (20 + 4\omega_2 + \omega_4)\mathbb{Z} + 21\mathcal{O}_K$ darstellen. □

Algebraische Zahlen können wir nun durch ihre Koeffizientenvektoren darstellen und die algebraischen Zahlen $a_1, \ldots, a_m$, die den Modul bzw. das Ideal erzeugen, können wir als Folge von Vektoren oder auch als Matrix auffassen. Sei $\{\omega_1, \ldots, \omega_n\}$ eine Basis von $\mathcal{O}$, dann können wir also einen Modul $M \subseteq \mathcal{O}$ durch eine Matrix $A \in \mathbb{Z}/d\mathbb{Z}^{n \times m}$ mit folgenden Eigenschaften darstellen:

1. $\exp(M)|d$.
2. Die algebraischen Zahlen $(\bar{\alpha}_1, \ldots, \bar{\alpha}_m) = (\bar{\omega}_1, \ldots, \bar{\omega}_n) \cdot A$ aus $M/d\mathcal{O}$ erzeugen $M/d\mathcal{O}$ als $\mathbb{Z}/d\mathbb{Z}$–Modul, d.h.

$$M = (A) = d\mathcal{O} + \alpha_1\mathbb{Z} + \cdots + \alpha_m\mathbb{Z},$$

wobei $\bar{\cdot}$ Reduktion modulo d bezeichnet.

Im folgenden werden wir jedoch bis auf die Beispiele einen solchen Modul durch das Paar (d, A) darstellen, damit man in der Notation den für unsere Betrachtungen sehr wichtigen Modul d sofort ablesen kann. In den Beispielen hingegen werden wir uns stärker an die Implementierung anlehnen und $(A \bmod d)$ schreiben, um deutlich zu machen, dass der Modul in der Klasse zur Darstellung von Matrizen über $\mathbb{Z}/d\mathbb{Z}$ versteckt ist.

3.1.3. Beispiel

In unserem Beispiel erhalten wir also

$$\mathfrak{a} = \left(\begin{pmatrix} 17 & 20 \\ 3 & 4 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \bmod 21 \right)$$

als Darstellung unseres Ideals. □

Mit dieser Darstellung erreichen wir, dass es unter Umständen möglich ist, eine Matrix mit weniger Spalten als zuvor anzugeben (in unserem Beispiel nun 2 statt 4). Zusätzlich benötigen wir in der Darstellung im allgemeinen Fall noch den Nenner a, wie wir ihn bereits oben erklärt haben.

In der Praxis werden an die Darstellung zwei widersprüchliche Anforderungen gestellt. Einerseits bevorzugen wir im Hinblick auf den Gleichheitstest eine eindeutige Darstellung. Dazu müssten wir stets den exakten Exponenten eines Moduls bzw. eines Ideals und nicht ein Vielfaches davon, sowie ein sog. normalisiertes Standard–Erzeugendensystem (vgl. Kapitel 3.3.3) zu seiner Darstellung verwenden. Demgegenüber benötigen wir aber eine schnelle Arithmetik, was bedeutet, dass wir evtl. nicht die Zeit zur Berechnung des Exponenten aufbringen wollen und darüber hinaus evtl. andere Erzeugendensysteme mit möglichst wenig Komponenten zur Darstellung verwenden wollen. Wie wir sehen werden, ist der Zeitbedarf der Algorithmen zur Idealarithmetik nämlich in erster Linie von der Größe der Erzeugendensysteme abhängig. Hier einen geeigneten Kompromiss zu finden, ist eine der Aufgaben einer praktischen Implementierung.

3.1.2 Addition und Durchschnitt

Seien $\mathfrak{a} = (l, A) = (l, (\underline{a}_1, \ldots, \underline{a}_{r_1}))$ und $\mathfrak{b} = (m, B) = (m, (\underline{b}_1, \ldots, \underline{b}_{r_2}))$ zwei $\mathbb{Z}$–Moduln, die in $\mathcal{O}$ liegen.

Dann ist klar, dass $\underbrace{\mathrm{ggT}(l, m)}_{=:q}\mathcal{O} \subseteq \mathfrak{a} + \mathfrak{b}$ und daher gilt auch:

$$\mathfrak{a} + \mathfrak{b} = (\mathfrak{a} + q\mathcal{O}) + (\mathfrak{b} + q\mathcal{O}) = (q, \overline{A}) + (q, \overline{B}),$$

wobei $\bar{\cdot}$ Reduktion modulo q bezeichnet. Damit ist die Berechnung der Summe zweier Moduln auf die Berechnung der Summe zweier Moduln in $\mathcal{O}/q\mathcal{O}$ bzw. über $(\mathbb{Z}/q\mathbb{Z})^n$ zurückgespielt. Diese Aufgabe kann mit dem Verfahren, das wir in Kapitel 3.3.7 vorstellen werden, gelöst werden.

3.1.4. Beispiel

Wir betrachten die Ideale

$$\mathfrak{a} = \left(\left(\begin{array}{cc} 17 & 20 \\ 3 & 4 \\ 1 & 0 \\ 0 & 1 \end{array} \right) \bmod 21 \right), \mathfrak{b} = \left(\left(\begin{array}{cc} 11 & 8 \\ 12 & 1 \\ 1 & 0 \\ 0 & 1 \end{array} \right) \bmod 15 \right).$$

Wegen $\mathrm{ggT}(21, 15) = 3$ ist dann $\mathfrak{a} + \mathfrak{b} = (3, C)$ wobei C die Summe der durch die reduzierten Erzeugendensysteme

$$\left(\begin{array}{cc} 2 & 2 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{array} \right) \bmod 3 \text{ und } \left(\begin{array}{cc} 2 & 2 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{array} \right) \bmod 3$$

gegebenen Moduln über $\mathbb{Z}/3\mathbb{Z}$ ist, also ergibt sich als Ergebnis:

$$\mathfrak{a} + \mathfrak{b} = \left(\left(\begin{array}{cc} 2 & 2 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{array} \right) \bmod 3 \right).$$

□

Mit einem sehr ähnlichen Trick können wir die Berechnung des Durchschnitts ebenfalls auf den Fall von Kapitel 3.3.7 zurückführen. Leider können wir nicht unmittelbar den Durchschnitt von Moduln berechnen, die in der Form

$$\mathfrak{a} = l\mathcal{O} + \underline{a}_1\mathbb{Z} + \cdots + \underline{a}_{r_1}\mathbb{Z}$$

dargestellt sind, da wir nicht wissen, wie man die Komponente $m\mathcal{O}$ bzw. $l\mathcal{O}$ der Darstellung behandeln soll. Aber es gilt $\mathrm{kgV}(l, m)\mathcal{O} \subseteq \mathfrak{a}, \mathfrak{b}, \mathfrak{a} \cap \mathfrak{b}$. Daher liften wir die Darstellungen von $\mathfrak{a}$ und $\mathfrak{b}$ zunächst nach $\mathcal{O}/(\mathrm{kgV}(l, m)\mathcal{O})$ und können dann unmittelbar die in Kapitel 3.3.7 beschriebene Methode anwenden.

Dieses Liften kann wie folgt stattfinden: Ist $\underline{a}_1, \ldots, \underline{a}_k$ ein Erzeugendensystem von $A \bmod l\mathcal{O}$ und $\underline{\omega}_1, \ldots, \underline{\omega}_n$ eine Basis von $\mathcal{O}$, so ist $\underline{a}_1, \ldots, \underline{a}_k, l\underline{\omega}_1, \ldots, l\underline{\omega}_n$ ein Erzeugendensystem von $A \bmod \mathrm{kgV}(l, m)\mathcal{O}$. Dieses kann mit den Methoden aus Kapitel 3.3 zu einem Erzeugendensystem mit n Komponenten verkleinert werden.

Wir man sieht ist unsere Ideal–Darstellung für die Berechnung des Durchschnitts vergleichsweise ineffizient, da der Lifting-Schritt fast darauf hinausläuft, doch wieder eine $\mathbb{Z}$–Basis zu ermitteln, allerdings ist zu beachten, das die Berechnung des Durchschnitts eine nur sehr selten verwendete Operation ist, so dass diese Ineffizienz für unsere Belange keine Rolle spielt.

3.1.3 Multiplikation

Die Multiplikation wollen wir nur für Ideale betrachten, da dadurch ein einfacherer und effizienterer Algorithmus ermöglicht wird.

Dieser beruht auf dem folgenden Resultat:

3.1.5. Proposition
Seien $\mathfrak{a} = (l, A) = (l, (\underline{a}_1, \ldots, \underline{a}_{r_1}))$ *und* $\mathfrak{b} = (m, B) = (m, (\underline{b}_1, \ldots, \underline{b}_{r_2}))$ *zwei ganze Ideale. Dann ist ihr Produkt gegeben durch*

$$(l \cdot m, (\underline{a}_1\underline{b}_1, \ldots, \underline{a}_1\underline{b}_{r_2}, \ldots, \underline{a}_{r_1}\underline{b}_1, \ldots, \underline{a}_{r_1}\underline{b}_{r_2}, m\underline{a}_1, \ldots, m\underline{a}_{r_1}, l\underline{b}_1, \ldots, l\underline{b}_{r_2})).$$

Beweis:
Sei $c \in \mathfrak{a} \cdot \mathfrak{b}$. Dann existieren ganze algebraische Zahlen x_0 und y_0 und ganze rationale Zahlen $x_1, \ldots, x_{r_1}, y_1, \ldots, y_{r_2}$ mit

$$c = (lx_0 + \sum_{i=1}^{r_1} x_i\underline{a}_i) \cdot (my_0 + \sum_{i=1}^{r_2} y_i\underline{b}_i).$$

Durch Ausmultiplizieren ergibt sich:

$$c = \underbrace{lmx_0y_0}_{\in l\cdot m\cdot\mathcal{O}} + \underbrace{\sum_{i=1}^{r_1}\sum_{j=1}^{r_2} x_iy_j\underline{a}_i\underline{b}_j}_{\in\langle\underline{a}_i\underline{b}_j\rangle_{1\le i\le r_1, 1\le j\le r_2}} + \sum_{i=1}^{r_1} my_0x_i\underline{a}_i + \sum_{i=1}^{r_2} lx_0y_i\underline{b}_i.$$

Da $\mathfrak{a}$ aber nach Voraussetzung ein Ideal ist, gilt jeweils für ein festes i: $y_0\underline{a}_i \in \mathfrak{a}$ und somit

$$y_0\underline{a}_i = l \cdot z_0 + \sum_{j=1}^{r_1} z_j\underline{a}_j \tag{3.1}$$

für eine geeignete ganze algebraische Zahl z_0 und geeignete rationale ganze Zahlen $z_1, \ldots, z_{r_1}$, und somit ist

$$my_0x_i\underline{a}_i = \underbrace{lmz_0x_i}_{\in l\cdot m\cdot\mathcal{O}} + \underbrace{\sum_{j=1}^{r_1} x_iz_jm\underline{a}_j}_{\in\langle m\underline{a}_1,\ldots,m\underline{a}_{r_1}\rangle}.$$

Analog gilt diese Rechnung auch für $\mathfrak{b}$ und es ergibt sich die Behauptung. □

3.1.6. Beispiel

Wir wählen $\mathfrak{a}$ und $\mathfrak{b}$ wieder wie in Beispiel 3.1.4. Dann gilt also:

$$\mathfrak{ab} = \left\langle \left(\begin{array}{cccccccc} 310 & 81 & 249 & 80 & 255 & 300 & 231 & 168 \\ 100 & 7 & 103 & 5 & 45 & 60 & 252 & 21 \\ 94 & 2 & 161 & 311 & 15 & 0 & 21 & 0 \\ 112 & 58 & 136 & 83 & 0 & 15 & 0 & 21 \end{array} \right) \bmod 315 \right\rangle .$$

Wie wir in Beispiel 3.3.9 sehen werden, vereinfacht sich dies zu

$$\mathfrak{ab} = \left\langle \left(\begin{array}{cc} 101 & 188 \\ 87 & 46 \\ 1 & 0 \\ 0 & 1 \end{array} \right) \bmod 315 \right\rangle .$$

□

In der praktischen Implementierung berechnen wir also das in Proposition 3.1.5 angegebene Erzeugendensystem und reduzieren dann mit den Methoden, die wir in Kapitel 3.3 zeigen werden, die Größe des Erzeugendensystems. Dabei sollte man beachten, dass die Berechnung des Erzeugendensystems dadurch vereinfacht wird, dass $r_1 + r_2$ Multiplikationen nur Multiplikationen von algebraischen Zahlen mit einer rationalen ganzen Zahl sind. Diese sind erheblich schneller auszuführen als Multiplikationen von zwei algebraischen Zahlen.

3.1.7. Bemerkung

Für Moduln wird die Multiplikation dadurch erschwert, dass (3.1) leider nicht gilt und daher das Erzeugendensystem zusätzlich noch um die Komponenten aus $\{m \cdot \underline{a}_k \cdot \underline{\omega}_j : 1 \leq k \leq r_1, 1 \leq j \leq n\}$ und $\{l \cdot \underline{b}_k \cdot \underline{\omega}_j : 1 \leq k \leq r_2, 1 \leq j \leq n\}$ vergrößert werden muss, wobei $\underline{\omega}_1, \ldots, \underline{\omega}_n$ eine $\mathbb{Z}$–Basis von $\mathcal{O}$ bezeichnet.

3.1.4 Division

Falls $\mathfrak{b}$ ein invertierbares $\mathcal{O}$–Ideal ist, gilt $\mathfrak{b}^{-1} = (\mathcal{O} : \mathfrak{b})$, d.h. $\mathfrak{b} \cdot (\mathcal{O} : \mathfrak{b}) = (\mathcal{O} : \mathfrak{b}) \cdot \mathfrak{b} = \mathcal{O}$. Im allgemeinen gilt aber nur $\mathfrak{b} \cdot (\mathcal{O} : \mathfrak{b}) = (\mathcal{O} : \mathfrak{b}) \cdot \mathfrak{b} \subseteq \mathcal{O}$. In diesem Fall ist aber $(\mathcal{O} : \mathfrak{b})$ die bestmögliche Approximation an ein Inverses. Falls $\mathfrak{b}$ ein $\mathcal{O}$–Ideal ist, aber $(\mathcal{O} : \mathfrak{b}) \cdot \mathfrak{b} \subsetneq \mathcal{O}$, können wir die Ordnung $\mathcal{O}$ leicht so maximieren, dass wir ein echtes Inverses in der größeren Ordnung bestimmen können: $(\mathfrak{b} : \mathfrak{b})$ ist der Multiplikatorenring von $\mathfrak{b}$ und wie man leicht zeigt die größere Ordnung, die uns interessiert. Hieran sehen wir, dass die Berechnung des

Multiplikatorenrings von besonderer Bedeutung ist. Tatsächlich lässt sich für diesen Spezialfall auch ein etwas effizienterer Algorithmus zur Division angeben.

Um einen effizienteren Algorithmus zur Division formulieren zu können, seien $\mathfrak{a}$ ein Ideal und $\mathfrak{b}$ ein Modul von vollem Rang. Die Berechnung des Pseudo–Quotienten basiert auf folgender Proposition:

3.1.8. Proposition
Seien $\mathfrak{a}$ ein ganzes $\mathcal{O}$–Ideal und $\mathfrak{b} \subset \mathcal{O}$ ein vollständiger Modul, so dass m ein Vielfaches von $\exp(\mathfrak{b})$ *ist. Dann gilt:*

$$m\mathfrak{a} \subseteq m(\mathfrak{a} : \mathfrak{b}) \subseteq \mathfrak{a}.$$

Beweis:
Da $\mathfrak{b} \subseteq \mathcal{O}$ und $\mathfrak{a}$ ein Ideal ist, ist auch $\mathfrak{a} \cdot \mathfrak{b} \subseteq \mathfrak{a} \cdot \mathcal{O} \subseteq \mathfrak{a}$, also gilt nach Definition von $(\mathfrak{a} : \mathfrak{b})$, dass $\mathfrak{a} \subseteq (\mathfrak{a} : \mathfrak{b})$ ist.

Um die zweite Inklusion zu zeigen, fixiere $x \in (\mathfrak{a} : \mathfrak{b})$. Wegen $\exp(\mathfrak{b}) \in \mathfrak{b}$ gilt $x \cdot \exp(\mathfrak{b}) \in \mathfrak{a}$ (nach Definition von $(\mathfrak{a} : \mathfrak{b})$). Daher gilt auch $x \in \frac{1}{\exp(\mathfrak{b})}\mathfrak{a}$ für alle $x \in (\mathfrak{a} : \mathfrak{b})$. Man beachte, dass $\frac{1}{\exp(\mathfrak{b})}$ wohldefiniert ist, da $\mathfrak{b}$ ein vollständiger Modul ist, d.h. $exp(\mathfrak{b}) > 0$. □

Der Divisionsalgorithmus beruht nun auf dem folgenden Resultat:

3.1.9. Satz
Sind $\mathfrak{a}$ ein Ideal und $\mathfrak{b} \subseteq \mathcal{O}$ ein vollständiger Modul, so dass m ein Vielfaches von $\exp(\mathfrak{b})$ *ist, dann ist $m \cdot (\mathfrak{a} : \mathfrak{b})$ der Kern des Homomorphismus*

$$\begin{array}{rcl} \phi : \ {}^{\mathfrak{a}}\!/_{m\mathfrak{a}} & \longrightarrow & \mathrm{Hom}\left({}^{\mathfrak{b}}\!/_{m\mathcal{O}} \to {}^{\mathfrak{a}}\!/_{m\mathfrak{a}} \right) \\ (\alpha + m\mathfrak{a}) & \longmapsto & (x + m\mathcal{O} \mapsto \alpha x + m\mathfrak{a}). \end{array}$$

Beweis:
Zunächst ist ϕ wohldefiniert, da

$$(\alpha + m\mathfrak{a}) \cdot (x + m\mathcal{O}) = \underbrace{\alpha \cdot x}_{\in \mathfrak{a}} + \underbrace{m\alpha}_{\in m\mathfrak{a}} \mathcal{O} + \underbrace{m\mathfrak{a}(x + m\mathcal{O})}_{\in m\mathfrak{a}} \in {}^{\mathfrak{a}}\!/_{m\mathfrak{a}}.$$

Dann sind äquivalent:

$$\begin{array}{rcll} \alpha + m\mathfrak{a} \in \mathrm{Kern}(\phi) & \Leftrightarrow & (\alpha + m\mathfrak{a}) \cdot (x + m\mathcal{O}) \in m \cdot \mathfrak{a} & \forall x \in \mathfrak{b} \\ & \Leftrightarrow & \alpha \cdot \mathfrak{b} \subseteq m \cdot \mathfrak{a}, \text{ nach Proposition 3.1.8} & \\ & \Leftrightarrow & \alpha \in ((m \cdot \mathfrak{a}) : \mathfrak{b}) = m(\mathfrak{a} : \mathfrak{b}). & \end{array}$$

□

Damit ist auch die Division auf die Lineare Algebra aus Kapitel 3.3 zurückgespielt. Da dieser Algorithmus jedoch relativ kompliziert ist, wollen wir ihn beispielhaft vollständig angeben:

3.1.10. Algorithmus

Idealdivision

EINGABE: Ein Ideal $\mathfrak{a}$ und ein vollständiger Modul $\mathfrak{b}$
AUSGABE: Der (Pseudo)quotient $(\mathfrak{a} : \mathfrak{b})$

(1) Berechne $\mathbb{Z}$–Basen A von $\mathfrak{a}$ bzw. B von $\mathfrak{b}$.
(2) **for** $(i = 1; i \leq n; i++)$ **do**
(3) **for** $(j = 1; j \leq n; j++)$ **do**
(4) Speichere die Darstellung von $\underline{a}_i \cdot \underline{b}_j$ als $((i-1)*n+j)$–te Spalte der Matrix C.
(5) **od**
(6) **od**
(7) Bestimme die n^2 Lösungen der n^2 Gleichungssysteme

$$A \cdot X = C.$$

// Jetzt stellen je n sukzessive Spalten von X das Bild eines Basis-
// vektors unter ϕ dar.
(8) Speichere je n aufeinanderfolgende Spalten von X als eine Spalte der $n^2 \times n$ Matrix M ab.
(9) Berechne mit Algorithmus 3.3.16 den Kern von M modulo m.
// Jetzt kennen wir den Kern bzgl. der Basis von $\mathfrak{a}$ und müssen ihn
// nun bzgl. der Basis von $\mathcal{O}$ darstellen. Dabei treten höchstens
// Zahlen der Größenordnung $l \cdot m$ auf.
(10) Lifte die Basis von $\mathfrak{a}$ und den Kern nach $\mathbb{Z}/lm\mathbb{Z}$ und multipliziere die beiden Matrizen zu einer Matrix P.
(11) Gib $1/m \cdot (lm, P)$ zurück.

An dieser Stellen wollen wir auch beispielhaft die Komplexität des Algorithmus betrachten: Zunächst berechnen wir die Darstellungsmatrix von ϕ (Schritte 1–4), Dazu berechnen wir $\mathbb{Z}$–Basen von $\mathfrak{a}$ und $\mathfrak{b}$, was $O(n^3)$ arithmetische

Operationen $\bmod l$ bzw. $\bmod m$ erfordert. Dann multiplizieren wir jedes Element der einen Basis mit jedem Element der anderen Basis. Dies kann $\bmod lm$ geschehen und erfordert $O(n^4)$ arithmetische Operationen $\bmod lm$. Die Ergebnisse all dieser Multiplikationen sind jeweils Elemente von $\mathfrak{a}$ und können daher relativ zur $\mathbb{Z}$–Basis von $\mathfrak{a}$ dargestellt werden. Dazu müssen wir gleichzeitig (!) n^2 Gleichungssysteme $\bmod lm$ lösen, wozu $O(n^4)$ arithmetische Operationen $\bmod lm$ ausreichen. Nun müssen wir den Kern einer $n^2 \times n$–Matrix über $\mathbb{Z}/m\mathbb{Z}$ berechnen. Laut Satz 3.3.17 sind hierzu $O(n^5)$ arithmetische Operationen mod m nötig. Um schließlich die Ergebnismatrix zu erhalten, multiplizieren wir die $\mathbb{Z}$–Basis von $\mathfrak{a}$ mit den Spalten des Kerns. Dies kann mit $O(n^3)$ Operationen modulo lm geschehen, da dies der Exponent des (Pseudo–) Quotienten ist.

3.1.11. Bemerkung
Falls $\mathfrak{a}$ kein Ideal ist, so wird die Berechnung dadurch erschwert, dass lediglich $\mathfrak{a} \cdot \mathfrak{b} \subseteq \mathcal{O} \cdot \mathcal{O} \subseteq \mathcal{O}$ gilt. Daher müssen wir nun den Homomorphismus

$$\phi : \mathfrak{a}/m\mathfrak{a} \longrightarrow \mathrm{Hom}\left(\mathfrak{b}/m\mathcal{O} \to \mathcal{O}/m\mathfrak{a}\right)$$

betrachten. Da wir die Matrix des Homomorphismus nicht mehr relativ zur Basis von $\mathfrak{a}$ darstellen und Reduktion mod m anwenden können, sondern statt dessen nun eine Reduktion mod $m\mathfrak{a}$ ausführen müssen, muss die originale $n^2 \times n$–Matrix durch eine $n^2 \times n^2$–Matrix erweitert werden, die in geeigneter Weise $m\mathfrak{a}$ darstellt, d.h. wir müssen nun den Kern einer $n^2 \times ((n+1)n)$–Matrix anstelle des Kerns einer $n^2 \times n$–Matrix berechnen.

3.1.5 Gleichheitstest

Seien $\mathfrak{a} = (l, A) = (l, (\underline{a}_1, \ldots, \underline{a}_{r_1}))$ und $\mathfrak{b} = (m, B) = (m, (\underline{b}_1, \ldots, \underline{b}_{r_2}))$ zwei $\mathbb{Z}$–Moduln, die in $\mathcal{O}$ liegen.

Da wir gestatteten, dass l und m beliebige Vielfache der Exponenten von $\mathfrak{a}$ und $\mathfrak{b}$ sind, und weil wir keine Eindeutigkeit der Matrizen A und B verlangen wollen, ist der Gleichheitstest nun relativ aufwändig. Da dieser in den von uns implementierten Algorithmen allerdings nur sehr selten vorkommt, ist dies kein Problem.

Falls $l = m$, dann sind $\mathfrak{a}$ und $\mathfrak{b}$ genau dann gleich, wenn A und B denselben Modul über $\mathbb{Z}/m\mathbb{Z}$ erzeugen. Gleichheit kann also dann so getestet werden, wie wir dies in Kapitel 3.3.5 beschreiben werden.

Andernfalls müssen wir testen, ob $\mathrm{ggT}(l, m)$ ein Vielfaches der Exponenten von $\mathfrak{a}$ und $\mathfrak{b}$ ist. Dies kann geschehen, indem wir testen, ob für $1 \leq i \leq n$

$\mathrm{ggT}(l,m)\cdot\underline{\omega}_i$ in $\mathfrak{a}$ und in $\mathfrak{b}$ liegt ($\underline{\omega}_1,\ldots,\underline{\omega}_n$ bezeichne eine $\mathbb{Z}$–Basis von $\mathcal{O}$). Falls diese Bedingung nicht erfüllt ist, ist $\mathfrak{a}$ verschieden von $\mathfrak{b}$. Falls sie erfüllt ist, können wir $\mathfrak{a}$ und $\mathfrak{b}$ mod $\mathrm{ggT}(l,m)$ reduzieren und das Verfahren für $l=m$ anwenden.

3.2 Implementierung von Moduln und Idealen

Wie gerade beschrieben stellen wir einen Modul bzw. ein Ideal, das in $\mathcal{O}$ liegt, für ein Vielfaches d des Exponenten mittels einer Matrix aus $\mathbb{Z}/d\mathbb{Z}$ dar. Daher besteht ein Objekt der Klasse `module` bzw. `alg_ideal` aus einem `nf_base *`, das auf die Basis der Ordnung $\mathcal{O}$ zeigt, bezüglich der der Modul gegeben ist, aus einer `bigmod_matrix base`, die d und die Matrix enthält, sowie einem Nenner `den`, damit auch Ideale bzw. Moduln verarbeitet werden können, die nicht in $\mathcal{O}$ liegen. Ferner enthält die Klasse einen boolschen Wert `is_exp`, der `true` ist, falls wir wissen, dass d der Exponent des Moduls ist. Auf dieser Darstellung aufbauend, haben wir die Algorithmen zur Idealarithmetik implementiert wie in Kapitel 3.1 beschrieben. Dabei besteht der Unterschied zwischen beiden Klassen darin, dass in der Klasse `module` die allgemeinen, langsamen Verfahren, die wir in Kapitel 3.1 nur angedeutet haben, für die Klasse `alg_ideal` hingegen die schnelleren, idealspezifischen Algorithmen implementiert wurden. Da dieser Unterschied nur für einzelne Operationen existiert, leiten wir die Klasse `alg_ideal` von der Klasse `module` ab und spezialisieren nur diese wenigen Funktionen.

Daneben benötigt man auch eine Darstellung der Basisvektoren mittels Konjugiertenvektoren, um die Reduktionstheorie anwenden zu können, bei der ein Element aus einem Ideal mit kurzem Konjugiertenvektor gefunden werden muss. Da dies jedoch der einzige Fall ist, wo wir diese Darstellung benötigen, die wiederum Präzisionsprobleme mit sich bringt, und da eine solche Reduktion für ein Ideal typischerweise höchstens einmal stattfindet, haben wir uns dafür entschieden, die Konjugiertenvektoren im Bedarfsfall aus den Konjugiertenvektoren der Basis der Ordnung zu berechnen. Damit erhalten wir die folgenden Klassendefinitionen:

```
class module{
protected:
  mutable bigmod_matrix base;
  bigint den;
```

```
  nf_base * O;
  mutable bool is_exp;

  void compare(const module&, bool &, bool &) const;
    // internal routine for comparisons

public:
  // Contructors & destructor:
  module(const nf_base * O1= nf_base::current_base);
  // zero module
  module(const alg_number & a,
         const alg_number & b = alg_number(bigint(0)));
  module(const base_matrix <bigint> &, const bigint & d = 1,
         const nf_base * O1 = nf_base::current_base);
  module(const bigmod_matrix &, const bigint & d = 1,
         const nf_base * O1 = nf_base::current_base);
  module(const module &);
  ~module();

  virtual module & operator =(const module & A);

  // member-functions
  const bigint & denominator() const;
  const bigmod_matrix & coeff_matrix() const;
  bigint_matrix z_basis() const;
  module numerator() const;
  nf_base * which_base() const;

  lidia_size_t degree() const;

  bool is_zero() const;         // In the obvious sense
  bool is_whole_order() const;
  bool is_one() const           // i.e. is_whole_order
  { return is_whole_order();}

  void normalize();
```

```
void invert();

void assign_zero();
// Remains in the same order

inline void assign_whole_order();
// Remains in the same order

void assign_one()
  // Remains in the same order, is the same as:
{assign_whole_order();}

void assign(const bigint &);
// Remains in the same order

void assign(const alg_number &);
// Becomes subset of same order as a !
virtual void assign(const module &);
// Becomes subset of same order as a !

// Procedural versions of arithmetic operations:
friend void add(module &, const module &, const module &);
friend void intersect(module &,
                      const module &, const module &);
friend void multiply(module &,
                     const module &, const module &);
friend void multiply(module &,
                     const module &, const bigint &);
friend void multiply(module &,
                     const bigint &, const module &);
friend void divide(module &,
                   const module &, const module &);
friend void divide(alg_ideal &,
                   const alg_ideal &, const module &);
friend void divide(module &,
```

```
                    const module &, const bigint &);
  friend void remainder(module &,
                        const module &, const bigint &);

  friend void power(module &, const module &, const bigint &);

  // arithmetic operators:
  friend module operator +(const module &,
                           const module &); // sum/union
  friend module operator &(const module &,
                           const module &); // intersection
  friend module operator *(const module &,
                           const module &); // product
  friend module operator *(const module &,
                           const bigint &); // product
  friend module operator *(const bigint &,
                           const module &); // product
  friend module operator /(const module &,
                           const bigint &); // quotient
  friend module operator /(const module &,
                           const module &); // quotient (??):
  // Division by an order is only guaranteed to produce
  // correct results, if you are in the maximal order!!

  friend module operator %(const module &,
                           const bigint &); // Reduce mod p0

  module& operator +=(const module & a);
  module& operator &=(const module & a);
  module& operator *=(const module & a);
  module& operator *=(const bigint & a);
  module& operator /=(const bigint & a);
  module& operator /=(const module & a);
  module& operator %=(const bigint & a);

  // Comparisions:
```

```
// By now, only comparision of modules
// over the same order is implemented.
friend bool operator ==(const module&,
                        const module&);
friend bool operator !=(const module&,
                        const module&);
friend bool operator <=(const module&,
                        const module&);
friend bool operator <(const module&,
                       const module&);
friend bool operator >=(const module&,
                        const module&);
friend bool operator >(const module&,
                       const module&);

bool operator ! () const
{return is_zero();}

// Some number-theoretic function:
friend lidia_size_t degree(const module &);
friend bigrational norm(const module &);      // Norm
friend bigrational exponent(const module &);     // exponent
order ring_of_multipliers(const bigint &p) const;

// other functions:
friend void invert(module &, const module &);
friend module inverse (const module &);

friend const bigint & denominator(const module &);
friend const bigmod_matrix & coeff_matrix(const module &);
friend bigint_matrix z_basis(const module &);
friend module numerator(const module &);
friend nf_base * which_base(const module &);

friend void square(module &, const module &);
```

```
  friend void swap(module &, module &);

  // random numbers
  void randomize(const bigint &);

  // In-/Output:
  friend ostream& operator<<(ostream &, const module &);
  friend istream& operator>>(istream &, module &);

  // friends:
  friend void multiply(alg_ideal &,
                       const alg_ideal &, const alg_ideal &);
};

class alg_ideal: public module{
public:
  // Contructors & destructor:
  alg_ideal(const nf_base * O1= nf_base::current_base);
  // zero ideal
  alg_ideal(const bigint & a,
        const alg_number & b = alg_number(bigint(0)));
  alg_ideal(const alg_number & a,
        const alg_number & b = alg_number(bigint(0)));
  alg_ideal(const base_matrix <bigint> &, const bigint & d = 1,
        const nf_base * O1 = nf_base::current_base);
  alg_ideal(const bigmod_matrix &, const bigint & d = 1,
        const nf_base * O1 = nf_base::current_base);
  alg_ideal(const alg_ideal &);
  ~alg_ideal();

  module & operator =(const module & A);
  void assign(const module & A);
  alg_ideal & operator =(const alg_ideal & A);

  // member-functions
```

```
alg_ideal numerator() const;

// Procedural versions of arithmetic operations:
friend void multiply(alg_ideal &,
                     const alg_ideal &, const alg_ideal &);
friend void multiply(alg_ideal &,
                     const alg_ideal &, const alg_number &);
friend void multiply(alg_ideal &,
                     const alg_number &, const alg_ideal &);
friend void divide(alg_ideal &,
                   const alg_ideal &, const module &);
friend void divide(alg_ideal &,
                   const alg_ideal &, const alg_number &);
friend long ord(const prime_ideal &, const alg_ideal &);

// arithmetic operators:
friend alg_ideal
operator +(const alg_ideal &,
           const alg_ideal &); // sum or union
friend alg_ideal
operator &(const alg_ideal &,
           const alg_ideal &); // intersection
friend alg_ideal
operator *(const alg_ideal &,
           const alg_ideal &); // product
friend alg_ideal
operator *(const alg_ideal &,
           const bigint &); // product
friend alg_ideal
operator *(const bigint &,
           const alg_ideal &); // product
friend alg_ideal
operator *(const alg_ideal &,
           const alg_number &); // product
friend alg_ideal
operator *(const alg_number &,
```

```
            const alg_ideal &); // product
  friend alg_ideal
  operator /(const alg_ideal &,
            const bigint &); // quotient
  friend alg_ideal
  operator /(const alg_ideal &,
            const alg_number &); // quotient
  friend alg_ideal
  operator /(const alg_ideal &,
            const module &); // quotient (??):
  // Division by an order is only guaranteed to produce
  // correct results, if you are in the maximal order!!

  alg_ideal& operator +=(const alg_ideal & a);
  alg_ideal& operator &=(const alg_ideal & a);
  alg_ideal& operator *=(const alg_ideal & a);
  alg_ideal& operator *=(const bigint & a);
  alg_ideal& operator /=(const bigint & a);
  alg_ideal& operator /=(const module & a);
  alg_ideal& operator %=(const bigint & a);

  // reduction:
  void reduce(alg_number & divisor);
  alg_number reduce();
  friend alg_number reduce(alg_ideal & c,
                           const alg_ideal & a);

  // other functions:
  friend alg_ideal inverse (const alg_ideal &);
  friend alg_ideal numerator(const alg_ideal &);

  // random numbers
  void randomize(const bigint &);
};
```

Zusammenfassend ergibt sich für die bisher beschriebenen Klassen zur algebraischen Zahlentheorie die in Abbildung 3.1 gezeigte Struktur. Dabei sym-

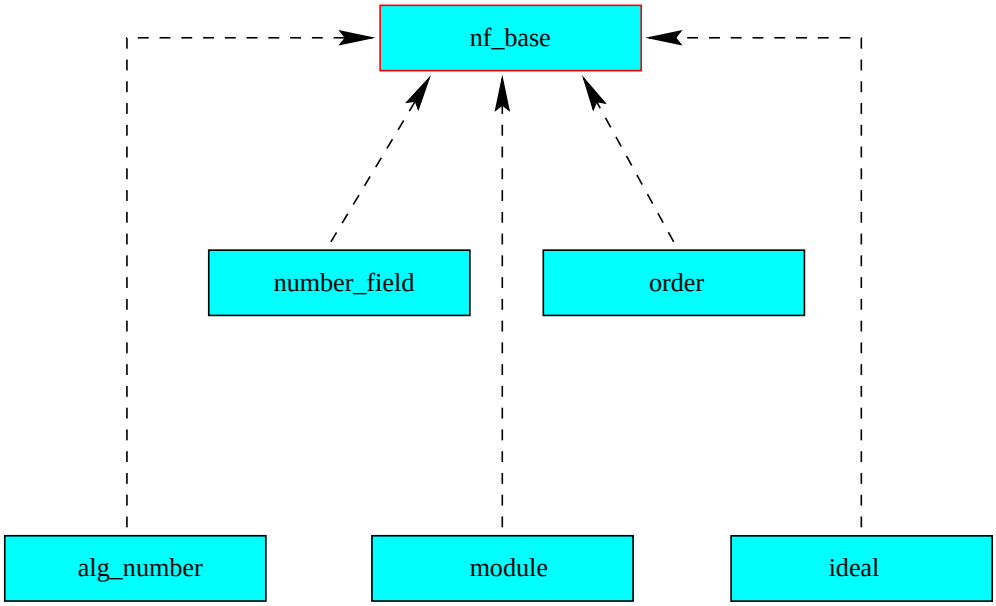

Abbildung 3.1: Die Struktur der Klassen für die algebraische Zahlentheorie.

bolisiert ein Pfeil von einer Klasse zu einer anderen Klasse die Tatsache, dass jedes Objekt der ersten Klasse einen Zeiger auf ein Objekt der zweiten Klasse enthält.

Bevor wir uns nun der Frage zuwenden, wie die benötigten Algorithmen der Linearen Algebra implementiert werden können, wollen wir zunächst einen Laufzeitvergleich geben, der zeigt, dass der gewählte Weg der Implementierung nicht nur wegen seiner Geschlossenheit theoretisch ansprechend ist, sondern dass er darüber hinaus auch auf Anhieb effizient implementiert werden kann.

Für den Laufzeitvergleich haben wir algebraische Zahlkörper verschiedener Körpergrade und Diskriminanten gewählt, wir betrachten allerdings stets Körper der Form $x^n + c$. Die Werte für n und c sind zusammen mit den Laufzeiten tabelliert. Dabei wurden die Laufzeiten auf einem Athlon–System mit 600 MHz unter OS/2 gemessen. Für die Addition fällt insbesondere auf, dass hier mit steigendem Körpergrad KaSh einen zunehmenden Laufzeitvorteil hat. Dies belegt, dass die Implementierung der Algorithmen zur Linearen Algebra über $\mathbb{Z}$ in KaSh deutlich ausgefeilter ist als unsere eher prototypische Implementierung der Algorithmen, die wir in Kapitel 3.3 beschreiben werden. Da jedoch die Idealarithmetik an der Laufzeit des von uns implementierten Verfahrens zur Klassengruppenberechnung nur einen geringen Anteil hat, erschien uns eine weitere Optimierung unserer Implementierung nicht notwendig. Diese

n	c	E	$T_{A,L}$	$T_{A,K}$	$T_{D,L}$	$T_{D,K}$	$T_{M,L}$	$T_{M,K}$
3	7	50	0,06s	0,06s	5,25s	3,07s	0,75s	1,32s
3	b	50	0,06s	0,06s	5,06s	3,22s	0,86s	1,36s
5	7	50	0,09s	0,07s	17s	19s	3,19s	4,19s
5	b	50	0,1s	0,08s	18s	19s	3,25s	4,26s
8	7	50	0,21s	0,11s	55s	94s	17s	14s
8	b	50	0,23s	0,13s	57s	109s	17s	11s

Tabelle 3.1: Laufzeiten zur Idealmultiplikation, -division und - addition in LiDIA und KaSh

n: Körpergrad, c: Polynomkoeffizient ($b = 12461287461287571241245 1$), E: Stellenzahl der Exponenten der Ideale, $T_{A,L/K}$: Zeit für 400 Idealadditionen mit LiDIA/KaSh, $T_{D,L/K}$: Zeit für 400 Idealdivisionen mit LiDIA/KaSh, $T_{M,L/K}$: Zeit für 400 Idealmultiplikationen mit LiDIA/KaSh.

wäre allerdings ohne weiteres möglich.

Hingegen zeigt die Idealdivision einen zunehmenden Vorteil für unsere Implementierung, wenn sie auch für kleinen Körpergrad zunächst langsamer ist. Schließlich zeigt sich bei der Idealmultiplikation in den absoluten Zahlen ein zunehmender Vorteil für KaSh. Allerdings können wir vermuten, dass wir hier mit einer entsprechend optimierten Implementierung für die Kern– und Bildberechnung in $\mathbb{Z}/m\mathbb{Z}$ auch einen deutlichen Vorteil unserer Implementierung sehen werden, da der Quotient aus Zeiten für die Multiplikation und Zeiten für die Addition, der den Zeitbedarf in Relation zum Zeitbedarf der Grundoperationen veranschaulicht, sich mit zunehmendem Körpergrad zu Gunsten unserer Implementierung verändert.

3.3 Lineare Algebra über Hauptidealringen

Nun wenden wir uns den Algorithmen der Linearen Algebra über $\mathbb{Z}/m\mathbb{Z}$ zu, die wir benötigen, um die bisher beschriebenen Aufgaben zu lösen. Dabei ist m wie gesehen eine möglicherweise zusammengesetzte ganze Zahl. Da sich die Algorithmen, die wir entwickelt haben, sofort auf den allgemeinen Fall der Hauptidealringe übertragen, wollen wir diese auch in voller Allgemeinheit darstellen.

Es sei also R im folgenden ein Hauptidealring, das heißt ein kommutativer Ring mit 1, in dem jedes Ideal ein Hauptideal ist, der jedoch *nicht notwendig nullteilerfrei* ist. Man beachte, dass im allgemeinen Sprachgebrauch „Haupt-

idealring“ häufig im Sinne von „Integritätsbereich, in dem jedes Ideal Hauptideal ist“ gebraucht wird. Einen Integritätsbereich verlangen wir hier jedoch ausdrücklich nicht.

Es seien l und k natürliche Zahlen. Wir werden zeigen, wie man die folgenden Aufgaben lösen kann:

1. Entscheide, ob ein Element $\underline{b} \in R^k$ Element eines gegebenen Teilmoduls von R^k ist.
2. Entscheide, ob ein Teilmodul von R^k in einem anderen derartigen Teilmodul enthalten ist.
3. Entscheide, ob zwei Teilmoduln von R^k gleich sind.
4. Berechne Kern und Bild des Homomorphismus $\varphi : R^l \longrightarrow R^k$. Berechne zu $\underline{b} \in R^k$ das Urbild $\varphi^{-1}(\{\underline{b}\}) = \{\underline{a} \in R^l : \varphi(\underline{a}) = \underline{b}\}$.

Ist R ein Körper, so lassen sich diese Probleme mittels Gaußelimination lösen, für nullteilerfreie Hauptidealringe können wir die Hermite–Normalform–Berechnung benutzen. Wir werden nun zeigen, wie man diese Verfahren modifizieren kann, um auch im Falle, dass der Ring Nullteiler enthält, zum Ziel zu kommen.

3.3.1 Das Berechnungsmodell

Wir setzen voraus, dass wir Algorithmen für die folgenden Grundoperationen zur Verfügung haben. Gegeben zwei Elemente $a, b \in R$.

- Entscheide, ob $a = b$.
- Berechne $a + b$, $a \cdot b$.
- Entscheide, ob $ax = b$ lösbar ist für ein $x \in R$ und falls ja, finde eine Lösung. Diese Operation werden wir *Division* in R nennen. In den Algorithmen werden wir die Funktion $\mathrm{div}(a, b)$ benutzen, die ein x zurückgibt, falls ein solches existiert und *error* sonst.
- Berechne den Erzeuger des Annihilators $\{x \in R : ax = 0\}$ von a. In den Algorithmen werden wir dazu die Funktion $\mathrm{an}(a)$ benutzen.
- Berechne g, x, y, e, f, so daß g das von a und b erzeugte Ideal erzeugt und

$$\begin{pmatrix} a & b \end{pmatrix} \begin{pmatrix} f & x \\ -e & y \end{pmatrix} = \begin{pmatrix} 0 & g \end{pmatrix} \text{ mit } ex + fy = 1.$$

Nach [OZ58], Kapitel IV, Theorem 33 gibt es solche g, x, y, e, f. Diese Operation werden wir *erweiterte* ggT-*Berechnung* nennen und in Algorithmen mit xgcdc(x, y, a, b, e, f) notieren.

In den Komplexitätsbetrachtungen werden wir *Ringoperationen* zählen, wobei jede der obigen Operationen (außer dem Vergleich) als eine *Ringoperation* gezählt wird. Vergleiche werden wir nicht mitzählen, da diese in der Praxis vergleichsweise sehr billig sind.

Die Berechnungen – insbesondere Vergleiche von Moduln – werden leichter, falls man Standarderzeuger von Idealen in R definieren kann, die durch das Ideal eindeutig bestimmt und leicht aus einem beliebigen Erzeuger berechenbar sind. Außerdem ist es nützlich, Standardvertreter von Restklassen modulo R-Idealen definieren zu können, die durch die Restklasse eindeutig bestimmt und leicht aus einem beliebigen Vertreter zu berechnen sind. Solche Standarderzeuger bzw. –vertreter zu berechnen, werden wir ebenfalls als eine Ringoperation zählen.

3.3.2 Zwei Beispiele

Standarderzeuger bzw. –vertreter sind über $\mathbb{Z}$ und $\mathbb{Z}/m\mathbb{Z}$ leicht zu definieren und auch zu berechnen:

$R = \mathbb{Z}$

Wir setzen die Realisierung arithmetischer Operationen in $\mathbb{Z}$ als bekannt voraus. Jedes $\mathbb{Z}$-Ideal hat genau einen nichtnegativen Erzeuger. Dieser kann als Standarderzeuger gewählt werden. Wenn n eine natürliche Zahl ist, dann hat jede Restklasse mod n genau einen Vertreter a mit $0 \leq a < n$. Dieser kann als Standardvertreter der Restklasse gewählt werden. Er kann mittels Division mit Rest leicht aus einem beliebigen Vertreter der Restklasse berechnet werden. Da $\mathbb{Z}$ nullteilerfrei ist, ist der Annihilator eines jeden von (0) verschiedenen Ideals (0). Die Funktion xgcdc kann mittels eines Algorithmus zur Berechnung des erweiterten ggT (etwa mit Hilfe des erweiterten euklidischen Algorithmus) leicht in der benötigten Form realisiert werden.

$R = \mathbb{Z}/m\mathbb{Z}$

Sei m eine natürliche Zahl. Dann können Addition und Multiplikation in $\mathbb{Z}/m\mathbb{Z}$ mittels Addition, Multiplikation und Division mit Rest in $\mathbb{Z}$ realisiert werden.

Für jedes von (0) verschiedene Ideal von $\mathbb{Z}/m\mathbb{Z}$ gibt es einen eindeutig bestimmten positiven Teiler von m, dessen Restklasse das Ideal erzeugt. Diesen Erzeuger benutzen wir als Standarderzeuger. Der Standarderzeuger des Ideals $(a + m\mathbb{Z})$ ist dann $\mathrm{ggT}(a, m) + m\mathbb{Z}$.

Ist nun a eine ganze Zahl und n ein Teiler von m. Als Standardvertreter der Restklasse von $a + m\mathbb{Z}$ modulo dem Ideal $I = (n + m\mathbb{Z})$ können wir $r + m\mathbb{Z}$ wählen, wobei r der kleinste nichtnegative Rest von a bei Division durch n ist. Dieser Vertreter ist in der Tat durch a und I eindeutig bestimmt.

Ist a eine ganze Zahl, dann besteht der Annihilator von $a + m\mathbb{Z}$ aus den Restklassen $x + m\mathbb{Z}$ mit $ax \equiv 0 \bmod m$, also wird der Annihilator erzeugt von $m/\mathrm{ggT}(a, m)$ (und dies ist bereits der Standarderzeuger).

Schließlich wollen wir $\mathrm{xgcdc}(x, y, a, b, e, f)$ so implementieren, dass wir gleich den Standarderzeuger $g + m\mathbb{Z}$ mit $g = \mathrm{ggT}(a, b, m)$ erhalten. Dazu benötigen wir folgendes Lemma:

3.3.1. Lemma *Für jedes Element $0 \neq a \in \mathbb{Z}/m\mathbb{Z}$, gibt es ein Element $x \in (\mathbb{Z}/m\mathbb{Z})^*$, so dass gilt: $x \cdot a | m$.*

3.3.2. Beispiel Zunächst wollen wir an einem Beispiel illustrieren, worin die eigentliche Aussage dieses Lemmas besteht. Wenn wir $a = 18$ und $m = 60$ wählen, dann gilt offensichtlich: $\mathrm{ggT}(18, 60) = 6$ und z.B. $(-3) \cdot 18 = 6|60$, jedoch ist $-3 \notin (\mathbb{Z}/60\mathbb{Z})^*$. Es gilt aber auch z.B. $7 \cdot 18 = 6|60$ und hier gilt: $7 \in (\mathbb{Z}/60\mathbb{Z})^*$.

Beweis: Sei allgemein $a \in \mathbb{Z}/m\mathbb{Z}$. Da $\mathbb{Z}$ ein Hauptidealring ist, gibt es x_0, y_0 mit

$$\begin{aligned} d := \mathrm{ggT}(a, m) &= x_0 a + y_0 m \\ &= (x_0 + k\frac{m}{d})a + (y_0 - k\frac{a}{d})m \\ &\equiv (x_0 + k\frac{m}{d})a \bmod m \quad (\forall k \in \mathbb{Z}) \end{aligned}$$

Dabei gilt $\mathrm{ggT}(x_0, \frac{m}{d}) = 1$, da $d \cdot \mathrm{ggT}(x_0, \frac{m}{d})$ ein Teiler von d ist, also ist $\mathrm{ggT}(x_0 + k\frac{m}{d}, \frac{m}{d}) = 1 \quad \forall k \in \mathbb{Z}$.

Wir zerlegen nun m in zwei Faktoren m_1 und m_2, so dass gilt:

- $m = m_1 \cdot m_2$.
- $\mathrm{ggT}(m_1, \frac{m}{d}) = 1$.
- m_2 enthält genau die Primfaktoren, die in $\frac{m}{d}$ vorkommen.

Im obigen Beispiel wäre also $m_1 = 3$ und $m_2 = 20$. Dann gilt: $m_1|d$ und $\frac{m}{d}|m_2$. Damit gilt auch $\mathrm{ggT}(x_0 + k\frac{m}{d}, m_2) = 1 \quad \forall k \in \mathbb{Z}$. Daher genügt es, zu zeigen, dass $\frac{m}{d}$ ein Erzeuger von $(\mathbb{Z}/m_1\mathbb{Z}, +)$ ist, denn dann können wir k_0 so wählen, dass gilt: $x_0 + k_0\frac{m}{d} \equiv 1 \bmod m_1$ und wir erhalten mit dem chinesischen Restsatz: $x := x_0 + k_0\frac{m}{d} \equiv 1 \bmod m$.

Dieser letzte Schritt gelingt wie folgt: Aus der Definition von m_1 folgt $\mathrm{ggT}(m_1, \frac{m}{d}) = 1$, also gibt es ein y mit $y\frac{m}{d} \equiv 1 \bmod m_1$ und 1 ist ein Erzeuger von $\mathbb{Z}/m_1\mathbb{Z}$, also muss auch $\frac{m}{d}$ ein Erzeuger dieser Gruppe sein. □

Damit funktioniert die Berechnung von $\mathrm{xgcdc}(x, y, a, b, e, f)$ nun wie folgt: Zunächst berechnen wir mit dem erweiterten euklidischen Algorithmus u und v mit

$$ua + vb = \mathrm{ggT}(a, b).$$

Dann ist

$$(ua + vb)/g = \mathrm{ggT}(a, b)/g.$$

Mit Lemma 3.3.1 berechnen wir nun eine ganze Zahl z mit

$$z\,\mathrm{ggT}(a, b)/g \equiv 1 \bmod m$$

und setzen

$$x = zu, y = zv, e = a/g, f = b/g.$$

Damit erhalten wir

$$xa + yb \equiv g \bmod m, \quad xe + yf \equiv 1 \bmod m.$$

3.3.3 Standard–Erzeugendensysteme für Teilmoduln von R^k

Seien k eine natürliche Zahl und M ein Teilmodul von R^k.

3.3.3. Definition *Für $i \in \{0, \ldots, k\}$ sei*

$$M^{(i)} := M \cap \{\underline{b} \in R^k \text{ mit } b_{i+1} = b_{i+2} = \ldots = b_k = 0\}.$$

Sei

$$S := S(M) := \{i : 1 \le i \le k, M^{(i)} \neq M^{(i-1)}\}. \tag{3.2}$$

Unter einem Standard–Erzeugendensystem *von M verstehen wir ein Erzeugendensystem $A = (\underline{a}_i)_{i\in S}$ von M, so dass für alle $i \in S$ das Teilsystem $(\underline{a}_j)_{j\le i}$ ein Erzeugendensystem von $M^{(i)}$ ist.*

Angenommen, in R können wir Standarderzeuger von Idealen und Standardvertreter von Restklassen wie oben beschrieben definieren. Dann wollen wir annehmen, dass xgcdc jeweils einen solchen Standarderzeuger zurückgibt. Dann definieren wir:

3.3.4. Definition *Ein* normalisiertes Standard–Erzeugendensystem *ist ein Standard–Erzeugendensystem A von M, in dem für jedes $i \in S(M)$ der Eintrag $a_{i,i}$ der Standarderzeuger von $a_{i,i}R$ und für jedes $j \in S$, $j > i$ der Eintrag $a_{i,j}$ der Standardvertreter von $a_{i,j} + a_{i,i}R$ ist.*

3.3.5. Bemerkung Wenn wir A als Matrix interpretieren fällt der Begriff des Standard–Erzeugendensystems gerade mit dem Begriff der *vollständig reduzierten Spaltenstufenform* zusammen, wie er erstmals von J. Howell definiert wurde([How86]).

Es gilt offensichtlich

$$\{0\} = M^{(0)} \subset M^{(1)} \subset \ldots \subset M^{(k)} = M.$$

Falls R nullteilerfrei ist, ist ein Standard–Erzeugendensystem bis auf Normalisierung gerade die Hermite–Normalform einer Basis von M.

Wir zeigen nun, dass auch im allgemeinen Fall jeder Modul M ein Standard–Erzeugendensystem hat. Dazu beachte man, dass die Menge aller i-ten Einträge in den Elementen von $M^{(i)}$ ein R–Ideal bildet, dass wir mit $I^{(i)}$ bezeichnen.

3.3.6. Satz *Sei $A = (\underline{a}_i)_{i \in S}$ eine Folge von Vektoren mit $\underline{a}_i \in M^{(i)}$ für alle $i \in S$. Dann ist A genau dann ein Standard–Erzeugendensystem von M, wenn $a_{i,i}$ das Ideal $I^{(i)}$ aller i–ten Einträge von Elementen aus $M^{(i)}$ erzeugt.*

Beweis: Sei A ein Standard–Erzeugendensystem von M. Fixiere $i \in S$. Da $(\underline{a}_j)_{j \leq i}$ $M^{(i)}$ erzeugt und da $\underline{a}_j \in M^{(j)}$ für jedes $j \in S$, folgt, dass jeder i-te Eintrag eines Elements aus $M^{(i)}$ ein Vielfaches von $a_{i,i}$ ist, also ist $a_{i,i}$ ein Erzeuger von $I^{(i)}$.

Für die umgekehrte Richtung setzen wir voraus, dass für alle $i \in S$ $a_{i,i}$ ein Erzeuger von $I^{(i)}$ ist. Wir zeigen mittels vollständiger Induktion, dass $(\underline{a}_j)_{j \leq i}$ ein Erzeugendensystem von $M^{(i)}$ ist. Für $i = 0$ ist dies trivial.

Sei $i \in \{1, \ldots, k\}$. Nach Induktionsvoraussetzung erzeugt dann $(\underline{a}_j)_{j \leq i'}$ $M^{(i')}$ für alle $i' < i$. Wenn i nicht in S liegt, ist also $M^{(i)} = M^{(i-1)}$ und die Behauptung folgt unmittelbar aus der Induktionsvoraussetzung.

Andernfalls ist $i \in S$. Falls i das kleinste Element von S ist, dann setze $i' = 0$, andernfalls setze i' auf das größte Element von S, das gerade noch kleiner ist als i. Sei $\underline{b}$ ein beliebiges Element aus $M^{(i)}$. Da $a_{i,i}$ das Ideal aller i-ten Einträge erzeugt, folgt, dass es ein $r \in R$ gibt mit $b_i = ra_{i,i}$. Dann ist $\underline{b} - r\underline{a}_i$ ein Element von $M^{(i)}$ dessen i-ter Eintrag 0 ist. Da $M^{(i')} = M^{(i'+1)} = \ldots = M^{(i-1)}$, folgt nun: $\underline{b} - r\underline{a}_j \in M^{(i')}$. Damit ist dieses Element eine Linearkombination der $\underline{a}_j$ mit $j \leq i'$. □

Der folgende Algorithmus formuliert explizit die von J. Howell in [How86] vorgestellte Idee und berechnet ein Standard–Erzeugendensystem eines Moduls M.

3.3.7. Algorithmus

Standard–Erzeugendensystem

EINGABE: Ein Erzeugendensystem $B \in R^{k \times l}$ eines Teilmoduls $M \subset R^k$
AUSGABE: Ein Standard–Erzeugendensystem A von M

```
(1)  for (i = k; i > 0; i − −) do
(2)    j = l
(3)    while (j > 0 ∧ b_ij = 0) do
(4)      j − −
(5)    od
(6)    if (j ≠ 0) then
(7)      vertausche die Spalten j und l von B;
(8)      for (j = l − 1; j > 0; j − −) do
(9)        if (b_ij ≠ 0) then
(10)         g = xgcdc(x, y, b_il, b_ij, e, f);
(11)         (b_l, b_j) = (x b_l + y b_j, f b_l − e b_j);
(12)       fi
(13)     od
```

```
(14)        a_i = b_l;
(15)        x = an(b_il);
(16)        if (x = 0) then
(17)           l − −
(18)        else
(19)           b_l = x b_l
(20)        fi
(21)     fi
(22)  od
```

3.3.8. Satz *Algorithmus 3.3.7 bestimmt ein Standard–Erzeugendensystem des Moduls M und benötigt dazu höchstens $6(k+1)kl$ Ringoperationen.*

Beweis: Es sei $B^{(k)} = B$ und für $1 \leq i \leq k$ bezeichne $B^{(i-1)}$ die Matrix B in Algorithmus 3.3.7 nachdem der Algorithmus die Iteration mit Index i beendet hat. Mit Induktion zeigen wir, dass $B^{(i)}$ $M^{(i)}$erzeugt, dass ein neues Element $\underline{a}_i$ genau dann zu A hinzugefügt wird, wenn $i \in S$ und dass in diesem Fall $a_{i,i}$ das Ideal aller i-ten Einträge von Elementen in $M^{(i)}$ erzeugt. Dann folgt mit Satz 3.3.6, dass A ein Standard–Erzeugendensystem von M ist.

Für $i = k$ ist die Behauptung wahr, da $B = B^{(k)}$ $M = M^{(k)}$ erzeugt.

Sei $i \in \{1, \ldots, k\}$. Nach Induktionsannahme erzeugt $B^{(i)}$ $M^{(i)}$. Die Zeilen mit den Indizes $i+1, \ldots, k$ in $B^{(i)}$ sind Null. In der nächsten Iteration eliminiert der Algorithmus die i-te Zeile.

Falls die i-te Zeile von $B^{(i)}$ nur Nulleinträge hat, dann sind nach Definition die Einträge in allen Elementen von $M^{(i)}$ Null. Also ist $M^{(i-1)} = M^{(i)}$ und die while–Schleife in den Schritten (3) bis (5) ergibt $j = 0$. Damit werden die Schritte (7) bis (21) nicht ausgeführt und es ist $B^{(i-1)} = B^{(i)}$. In diesem Falle erzeugt also $B^{(i-1)}$ $M^{(i-1)}$ und es wird kein Element zu A hinzugefügt.

Andernfalls ist $M^{(i)} \neq M^{(i-1)}$. Sei B' die Matrix, die aus $B^{(i)}$ nach Schritt

(14) entstanden ist. Diese Matrix ist von der Gestalt:

$$B' := \begin{pmatrix} b'_{11} & \dots & b'_{1l-1} & b'_{1l} \\ \vdots & & \vdots & \vdots \\ b'_{i-11} & \dots & b'_{i-1l-1} & b'_{i-1l} \\ 0 & \dots & 0 & b'_{il} \\ 0 & \dots & 0 & 0 \\ \vdots & & \vdots & \vdots \\ 0 & \dots & 0 & 0 \end{pmatrix}$$

und erzeugt immer noch $M^{(i)}$, da nur Transformationen mit Determinante ± 1 auf $B^{(i)}$ ausgeführt wurden, um B' zu erhalten. Außerdem ist $b'_{i,l}$ der einzige Eintrag verschieden von Null in der i-ten Zeile von B'. Daher erzeugt $b'_{i,l}$ das Ideal aller i-ten Einträge in Elementen von $M^{(i)}$. Da $\underline{a}_i$ auf $\underline{b}'_l$ gesetzt wird, folgt, dass $a_{i,i}$ das Ideal aller i-ten Einträge von Elementen in $M^{(i)}$ erzeugt. Schließlich ist jedes Element $\underline{b}$ in $M^{(i-1)}$ von der Form

$$\underline{b} = x_1\underline{b}'_1 + \dots + x_l\underline{b}'_l,$$

mit $x_i \in R$, $1 \le i \le l-1$ und x_l ist Element des Annihilators von b'_{il}. Wenn wir also $\underline{b}'_l$ durch $\mathrm{an}(b'_{i,l})\underline{b}'_l$ ersetzen, dann ist B' ein Erzeugendensystem von $M^{(i-1)}$. Damit ist die Korrektheit des Algorithmus bewiesen.

Nun untersuchen wir die Komplexität des Algorithmus. In der inneren Schleife (8) – (14) werden höchstens eine erweiterte ggT–Berechnung, $4k$ Multiplikationen und $2k$ Additionen in R ausgeführt, also insgesamt höchstens $6k+1$ Ringoperationen. Diese Schleife wird höchstens $(l-1)$–mal ausgeführt, womit sich maximal $(l-1)(6k+1)$ Ringoperationen ergeben. Außerdem werden in den Schritten (16) – (21) maximal $k+1$ Ringoperationen ausgeführt, also ergeben sich $(l-1)(6k+1)+k+1 < 6(k+1)l$ Ringoperationen für einen Durchlauf der äußeren Schleife. Diese wird k–mal ausgeführt, womit die Gesamtzahl der Operationen durch $6(k+1)kl$ beschränkt ist. □

3.3.9. Beispiel

Wir betrachten den Modul $M \subset (\mathbb{Z}/315\mathbb{Z})^4$ mit Erzeugendensystem

$$B = \begin{pmatrix} 310 & 81 & 249 & 80 & 255 & 300 & 231 & 168 \\ 100 & 7 & 103 & 5 & 45 & 60 & 252 & 21 \\ 94 & 2 & 161 & 311 & 15 & 0 & 21 & 0 \\ 112 & 58 & 136 & 83 & 0 & 15 & 0 & 21 \end{pmatrix} \bmod 315$$

und konstruieren gemäß Algorithmus 3.3.7 ein Standard–Erzeugendensystem. In der praktischen Implementierung verwenden wir eine etwas kompliziertere Pivotstrategie als in der Beschreibung des Algorithmus angegeben. Diese macht sich zunutze, dass 83 modulo 315 invertierbar ist und transformiert B durch Multiplikation der vierten Spalte mit $167 \in (\mathbb{Z}/315\mathbb{Z})^*$ und Vertauschen der vierten mit der letzten Spalte zunächst zu

$$\begin{pmatrix} 310 & 81 & 249 & 255 & 300 & 231 & 168 & 130 \\ 100 & 7 & 103 & 45 & 60 & 252 & 21 & 205 \\ 94 & 2 & 161 & 15 & 0 & 21 & 0 & 277 \\ 112 & 58 & 136 & 0 & 15 & 0 & 21 & 1 \end{pmatrix} \bmod 315.$$

Für $i = 4$ wird dann die Schleife (8)–(13) mit $l = 8$ ausgeführt und ergibt

$$\begin{pmatrix} 240 & 101 & 209 & 255 & 240 & 231 & 273 & 130 \\ 135 & 87 & 258 & 45 & 135 & 252 & 126 & 205 \\ 255 & 1 & 289 & 15 & 255 & 21 & 168 & 277 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \bmod 315.$$

Dann wird $\underline{a}_4 = (130, 205, 277, 1)^T$ gespeichert und da der Annihilator von 1 von 0 erzeugt wird, wird l auf 7 herabgesetzt. In der nächsten Iteration erkennt die implementierte Pivotstrategie 289 als invertierbares Element und vertauscht daher die Spalten 3 und 7 und multipliziert Spalte 3 dabei mit $109 \in (\mathbb{Z}/315\mathbb{Z})^*$. Für $i = 3$ wird dann die Schleife (8)–(13) mit $l = 7$ ausgeführt und ergibt

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 101 & 130 \\ 0 & 0 & 0 & 0 & 0 & 0 & 87 & 205 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 277 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \bmod 315.$$

Wir speichern $\underline{a}_3 = (101, 87, 1, 0)^T$ und stellen fest, dass wir fertig sind. Ein Standard–Erzeugendensystem für M ist also

$$A = \begin{pmatrix} 101 & 130 \\ 87 & 205 \\ 1 & 277 \\ 0 & 1 \end{pmatrix} \bmod 315.$$

□

Darüber hinaus können wir, falls der Ring dies zulässt (also z.B. im Falle von $\mathbb{Z}/m\mathbb{Z}$) mit folgender Prozedur ein normalisiertes Standard–Erzeugendensystem berechnen:

3.3.10. Algorithmus

Normalisierung

EINGABE: Ein Standard–Erzeugendensystem $A \in R^{k\times l}$ eines Moduls $M \subset R^k$
AUSGABE: Ein normalisiertes Standard–Erzeugendensystem A von M

(1) $S = S(M)$;
(2) **while** $(S \neq \emptyset)$ **do**
(3) $\quad i = \max(S)$;
(4) $\quad S' = S(M)$;
(5) $\quad$ **for** $(j = \max(S'); j > i; S' = S' \setminus \{j\})$ **do**
(6) $\quad\quad$ Sei r der Standardvertreter von $a_{i,j} + a_{i,i}R$;
(7) $\quad\quad \underline{a}_j = \underline{a}_j + \mathrm{div}(r - a_{i,j}, a_{i,i})\underline{a}_i$;
(8) $\quad$ **od**
(9) $\quad S = S \setminus \{i\}$;
(10) **od**

3.3.11. Satz *Es gibt genau ein normalisiertes Standard–Erzeugendensystem von M. Dieses kann mit Algorithmus 3.3.10 aus jedem Standard–Erzeugendensystem berechnet werden. Der Algorithmus benötigt dazu höchstens $k^3 + 1/2k^2$ Ringoperationen.*

Beweis: Mit Algorithmus 3.3.10 erhalten wir ein Standard–Erzeugendensystem mit den behaupteten Eigenschaften. Wir müssen noch zeigen, dass es kein anderes solches Erzeugendensystem geben kann. Angenommen B wäre ein zweites derartiges Erzeugendensystem. Fixiere $j \in S(M)$. Dann ist $a_{j,j} = b_{j,j}$, da nach Satz 3.3.6 diese Zahlen Standarderzeuger desselben Ideals sind. Sei $1 \leq i < j$ der größte Index mit $a_{i,j} \neq b_{i,j}$. Dann muss i in $S(M)$ liegen, da der Vektor $\underline{a}_j - \underline{b}_j$ in $M^{(i)}$, aber nicht in $M^{(i-1)}$ liegt. Dann gehören $a_{i,j}$ und $b_{i,j}$ zu verschieden Restklassen modulo $a_{i,i}$, da diese Einträge die Standardvertreter ihrer Restklassen sind. Also liegt $a_{i,j} - b_{i,j}$ nicht in $a_{i,i}R$. Dies widerspricht aber der Tatsache, dass nach Satz 3.3.6 der Eintrag $a_{i,i}$ das

Ideal aller i-ten Einträge von Elementen in $M^{(i)}$ erzeugt. Daraus folgt die Eindeutigkeit.

Wir schätzen noch den Aufwand für den Algorithmus ab: In der inneren Schleife (6) – (7) führt der Algorithmus höchstens $2k+3$ Ringoperationen aus. Diese Schleife wird höchstens $(k-i)$–mal für jedes $i \in S$ ausgeführt. Also werden höchstens $k(k-1)(2k+3)/2 < k^3 + k^2/2$ Ringoperationen ausgeführt. □

3.3.12. Beispiel

Das Standard–Erzeugendensystem aus Beispiel 3.3.9 wird durch Subtraktion des 277-fachen der ersten Spalte von der zweiten Spalte zu einem normalisierten Standard–Erzeugendensystem. Wir erhalten also

$$A = \begin{pmatrix} 101 & 188 \\ 87 & 46 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \bmod 315.$$

□

3.3.4 Element–Test

Wir zeigen nun, wie man mit Hilfe eines Standard–Erzeugendensystems entscheiden kann, ob ein Element $\underline{a} \in R^k$ zu einem Teilmodul $M \subset R^k$ gehört.

3.3.13. Algorithmus

Element–Test

EINGABE: Ein Standard–Erzeugendensystem $A \in R^{k \times s}$ eines Teilmoduls $M \subset R^k$ und $\underline{a} \in R^k$

AUSGABE: $(r_i)_{i \in S(M)}$ mit $\underline{a} = \sum_{i \in S(M)} r_i \underline{a}_i$ und $is\_element = true$, falls $\underline{a} \in M$. Andernfalls $is\_element = false$

(1) $S = S(M)$; $\underline{b} = \underline{a}$; $is\_element = false$
(2) **while** $(S \neq \emptyset)$ **do**
(3) $\quad i = \max(S)$;
(4) $\quad r_i = \mathrm{div}(b_i, a_{i,i})$;

```
(5)   if (r_i = error) then
(6)     return;
(7)   fi
(8)   b = b − r_i a_i;
(9)   S = S \ {i};
(10) od
(11) if (b = 0) then
(12)   (r_i)_{i∈S(M)}; is_element = true
(13) fi
```

3.3.14. Satz *Algorithmus 3.3.13 ist korrekt und benötigt höchstens $k(2k+1)$ Ringoperationen R.*

Beweis: Aus der Konstruktion, die der Algorithmus ausführt, ergibt sich, dass gilt

$$\underline{a} = \underline{b} + \sum_{j \geq i} r_j \underline{a}_j, \tag{3.3}$$

nachdem die While–Schleife für Index i abgearbeitet ist.

Falls der Algorithmus mit $\underline{b} = 0$ endet, so ist

$$\underline{a} = \sum_{i \in S(M)} r_i \underline{a}_i.$$

Wir müssen also zeigen, dass der Algorithmus immer mit $\underline{b} = 0$ endet, wenn $\underline{a}$ in M liegt.

Sei dazu $\underline{a} \in M$. Wir werden mit Induktion zeigen, dass $\underline{b} \in M^{(j)}$ mit $j = \max(S)$, falls $S \neq \emptyset$ und $j = 0$ sonst, bevor die Bedingung der While–Schleife geprüft wird. Dann gilt in jeder Iteration $r_i \neq error$. Daraus folgt, dass der Algorithmus mit $\underline{b} = 0$ endet.

Vor dem ersten Durchlauf ist $\underline{b} = \underline{a} \in M = M^{(\max(S(M)))}$. Angenommen, in Schritt (2) ist $S \neq \emptyset$ und $\underline{b}$ liegt in $M^{(\max(S))}$. Dann betritt der Algorithmus die While–Schleife und setzt i auf $\max(S)$. Nach Satz 3.3.6 muss b_i ein Vielfaches von $a_{i,i}$ sein. Dies wird in Schritt (4) erkannt, wo $r_i \neq error$ zurückgegeben wird. Der Vektor $\underline{b}$ und die Menge S werden dann so geändert, dass gilt: $\underline{b} \in M^{(\max S)}$, falls $S \neq \emptyset$ und $\underline{b} \in M^{(0)}$ sonst. Damit folgt die Korrektheit.

Nun untersuchen wir noch die Komplexität des Algorithmus. In Schritt (4) wird eine Division ausgeführt. In Schritt (8) werden k Multiplikationen

und k Subtraktionen ausgeführt. Also werden in jeder Iteration der While–Schleife $2k+1$ Ringoperationen ausgeführt. Diese Schleife wird für jedes Element des Standard–Erzeugendensystems von M einmal durchlaufen. Ein Standard–Erzeugendensystem von M hat höchstens k Elemente, also ergibt sich die behauptete Schranke. □

3.3.5 Tests auf Teilmenge und Gleichheit

Seien M und N Teilmoduln von R^k. Diese werden mittels Erzeugendensystemen mit höchstens k Elementen dargestellt. Wenn wir entscheiden wollen, ob N in M enthalten ist, können wir wie folgt vorgehen: Wir bestimmen ein Standard–Erzeugendensystem von M und für jedes Element des Erzeugendensystems von N prüfen wir, ob es in M liegt. Dies kann man wie eben beschrieben tun. Mit dem vorangegangenen Komplexitätsergebnis erhalten wir dann:

3.3.15. Satz *Man kann mit höchstens $k^2(8k+7)$ Ringoperationen testen, ob ein Modul $N \subset R^k$ in einem Modul $M \subset R^k$ enthalten ist. Falls $N \subset M$, gibt der Test außerdem die Darstellungen der Erzeuger von N als Linearkombinationen der Erzeuger von M an.*

Damit kann man einen Gleichheitstest realisieren, der höchstens $2k^2(8k+7)$ Ringoperationen benötigt. Wenn R allerdings eindeutige Idealerzeuger und eindeutige Vertreter von Restklassen zulässt und M und N mittels normalisierter Erzeugendensysteme gegeben sind, dann lässt sich Gleichheit einfach testen, indem man prüft, ob die darstellenden Erzeugendensysteme gleich sind.

3.3.6 Berechnung von Kern, Bild und Urbild

Sei

$$\varphi : R^l \to R^k$$

ein Homomorphismus, der durch eine Matrix $B \in R^{k\times l}$ gegeben ist. Dann wird das Bild von φ durch die Spalten von B erzeugt. Falls $l > k$, so kann man mit Algorithmus 3.3.7 ein Erzeugendensystem des Bildes mit höchstens k Elementen bestimmen.

Dabei kann Algorithmus 3.3.7 so modifiziert werden, dass wir kleinere Erzeugendensysteme erhalten. Z.B. können wir in Schritt (14) prüfen, ob das $\underline{b}_l$, das wir speichern wollen, ein Vielfaches eines zuvor gespeicherten $\underline{b}_l$ ist.

Andererseits können wir die Schritte (15), (16) und (18) – (20) einfach weglassen. Allerdings erhalten wir mit diesen Modifikationen im allgemeinen kein Standard–Erzeugendensystem mehr.

Um den Kern von φ zu bestimmen, müssen wir lediglich die Transformation T berechnen, die wir auf B in Algorithmus 3.3.7 anwenden. Um Urbilder berechnen zu können, speichern wir außerdem die Transformation U, die B nach A transformiert. Man beachte, dass die Transformationen T und U im allgemeinen verschieden sind. Der folgende Algorithmus berechnet Bild und Kern von φ. Dabei bezeichnet I_l die $l \times l$–Einheitsmatrix.

3.3.16. Algorithmus

Bild und Kern

EINGABE: Ein Homomorphismus $B \in R^{k \times l}$
AUSGABE: Das Bild A und der Kern T von B. Eine Transformation U mit $B \cdot U = A$

(1) **for** $(i = k;\ i > 0;\ i - -)$ **do**
(2) $\quad j = l;$
(3) $\quad T = I_l;$
(4) $\quad$ **while** $(j > 0 \wedge b_{ij} = 0)$ **do**
(5) $\quad\quad j - -;$
(6) $\quad$ **od**
(7) $\quad$ **if** $(j \neq 0)$ **then**
(8) $\quad\quad$ vertausche die Spalten j und l von B und von T;
(9) $\quad\quad$ **for** $(j = l - 1;\ j > 0;\ j - -)$ **do**
(10) $\quad\quad\quad$ **if** $(b_{ij} \neq 0)$ **then**
(11) $\quad\quad\quad\quad g = \mathrm{xgcdc}(x, y, b_{il}, b_{ij}, e, f);$
(12) $\quad\quad\quad\quad (\underline{b}_l, \underline{b}_j) = (x\underline{b}_l + y\underline{b}_j, f\underline{b}_l - e\underline{b}_j);$
(13) $\quad\quad\quad\quad (\underline{t}_l, \underline{t}_j) = (x\underline{t}_l + y\underline{t}_j, f\underline{t}_l - e\underline{t}_j);$
(14) $\quad\quad\quad$ **fi**
(15) $\quad\quad$ **od**
(16) $\quad\quad \underline{a}_i = \underline{b}_l;\ \underline{u}_i = \underline{t}_l;$
(17) $\quad\quad x = an(b_{il});$

```
(18)      if (x = 0) then
(19)        l − −
(20)      else
(21)        b_l = x b_l; t_l = x t_l;
(22)      fi
(23)    fi
(24)  od
(25)  Entferne alle Spalten mit Index > l aus B.
```

3.3.17. Satz *Algorithmus 3.3.16 ist korrekt und benötigt höchstens* $6(k+l)kl$ *Ringoperationen.*

Beweis: Aus Satz 3.3.8 wissen wir schon, dass Algorithmus 3.3.16 ein Standard–Erzeugendensystem des Bildes von φ berechnet. Setze $T^{(k)} = I_k$ und bezeichne mit $T^{(i-1)}$ die Matrix T, die man erhält, nachdem Algorithmus 3.3.16 die Iteration mit Index i beendet hat. Wie im Beweis von Satz 3.3.8 kann man mit Induktion zeigen, dass für $i = k, k-1, \ldots, 0$ die Spalten von $T^{(i)}$ den Modul aller $\underline{x} \in R^l$ mit $Bx \in M^{(i)}$ erzeugen. Wegen $M^{(0)} = \{0\}$ folgt dann sofort, dass die Matrix T, die man am Ende von Algorithmus 3.3.16 erhält, den Kern von B erzeugt. Genau wie in der Analyse von Algorithmus 3.3.7 sieht man, dass Algorithmus 3.3.16 höchstens $6(k+l)kl$ Ringoperationen benötigt. □

Zum Schluss diskutieren wir das Problem, Urbilder von Elementen aus R^k zu finden. Sei $\underline{b} \in R^k$. Um das Urbild von $\underline{b}$ unter der Abbildung φ zu berechnen, berechnen wir mit Algorithmus 3.3.16 den Kern und ein Standard–Erzeugendensystem des Bildes von φ. Dann wenden wir Algorithmus 3.3.13 an, um zu entscheiden, ob $\underline{b}$ im Bild von φ liegt. Falls nicht, so ist das Urbild von $\underline{b}$ leer. Andernfalls liefert Algorithmus 3.3.13 eine Lösung $\underline{r}$ von $A\underline{r} = \underline{b}$, wobei A das Standard–Erzeugendensystem des Bildes von φ ist. Aus Algorithmus 3.3.16 erhielten wir auch eine Transformation U mit $BU = A$. Also ist $BU\underline{r} = \underline{b}$, d.h. $U\underline{r}$ ist ein Urbild von $\underline{b}$. Das vollständige Urbild von $\underline{b}$ ist dann $\varphi^{-1}(\underline{b}) = U\underline{r} + \ker\varphi$. Die Berechnung von $U\underline{r}$ benötigt höchstens $(2k-1)l$ Ringoperationen. Mit Satz 3.3.17 und Satz 3.3.14 folgt, dass die gesamte Berechnung höchstens $6(k+l)kl + (k+l)(2k+1)$ Ringoperationen benötigt. Damit ist der folgende Satz bewiesen:

3.3.18. Satz *Um das Urbild von $\underline{b} \in R^k$ unter einem Homomorphismus $\varphi : R^l \to R^k$ zu berechnen, benötigt man höchstens $(k+l)(6lk+2k+1)$ Ringoperationen.*

3.3.7 Summe und Durchschnitt von Moduln

Seien M und N zwei R–Teilmoduln von R^k mit Erzeugendensystemen A bzw. B. Seien m die Zahl der Elemente von A und n die Zahl der Elemente von B. Die Summe von M und N wird über R von den Elementen von $C = A \circ B$, der Konkatenation von A und B erzeugt. Ein Erzeugendensystem von $M+N$ mit höchstens k Elementen kann unter Verwendung von Algorithmus 3.3.7 mit höchstens $6(k+1)k(m+n)$ Ringoperationen berechnet werden.

Wir beschreiben nun die Berechnung von $M \cap N$. Ein Element $\underline{a} \in R^k$ liegt genau dann in $M \cap N$, wenn es Vektoren $\underline{x} \in R^m$ und $\underline{y} \in R^n$ mit

$$A\underline{x} = \underline{a} = B\underline{y}$$

gibt.

Betrachte den Homomorphismus

$$\psi : R^m \times R^n \longrightarrow R^k, \quad (\underline{x}, \underline{y}) \longmapsto A\underline{x} - B\underline{y}.$$

Sei $((\underline{x}_1, \underline{y}_1), \ldots, (\underline{x}_l, \underline{y}_l))$ ein Erzeugendensystem des Kerns von ψ. Dann wird $M \cap N$ erzeugt von $(A\underline{x}_1, \ldots, A\underline{x}_l)$. Also berechnen wir mit Algorithmus 3.3.16 den Kern von ψ, um $M \cap N$ zu bestimmen. Dazu benötigen wir höchstens $6(k+m+n)k(m+n)$ Ringoperationen. Dann berechnen wir ein Erzeugendensystem von $M \cap N$ wie gerade beschrieben. Dazu sind höchstens $2km(m+n)$ Operationen nötig. Schließlich benutzen wir Algorithmus 3.3.7 um ein Erzeugendensystem von $M \cap N$ mit höchstens k Elementen zu finden. Dazu sind $6(k+1)k(m+n)$ Operationen ausreichend.

Insgesamt erhalten wir also folgende Resultate:

3.3.19. Satz *Die Summe von zwei Moduln $M, N \subset R^k$, die durch m bzw. n Erzeuger gegeben sind, kann mit höchstens $6(k+1)k(m+n)$ Operationen in R berechnet werden.*

3.3.20. Satz *Der Durchschnitt von zwei Moduln $M, N \subset R^k$, die durch m bzw. n Erzeuger gegeben sind, kann mit höchstens $(12k+8m+6n+6)k(m+n)$ Operationen in R berechnet werden.*

Kapitel 4

Primideale und Idealfaktorisierung

In diesem Kapitel beschreiben wir die besonderen Eigenschaften von Primidealen und zeigen die Implementierung der Klasse `prime_ideal`, die im Vergleich zur Klasse `alg_ideal` einige Überraschungen bietet. Danach wenden wir uns der Zerlegung von (gebrochenen) Idealen in ein Produkt von Primidealen zu und zeigen, wie wir diese Berechnungen mit Hilfe der allgemeinen Template–Klassen `factorization` $<$ `T` $>$ und `single_factor` $<$ `T` $>$ implementiert haben.

4.1 Primideale

Prinzipiell gibt es noch Alternativen zu der beschriebenen Darstellung von Moduln und Idealen, etwa die Darstellung mittels einer $\mathbb{Z}$–Basis, die wir in Kapitel 3 bereits verworfen haben. Insbesondere ist für den Fall der Ideale die sogenannte Zwei–Element–Darstellung von Bedeutung, wie sie in [PZ89], Kapitel 6.3 beschrieben wird. Diese beruht darauf, Ideale eben nicht als $\mathbb{Z}$–Moduln aufzufassen, sondern sie wirklich als Ideale über der Ordnung $\mathcal{O}$ zu betrachten, bezüglich der sie gegeben sind. Diese Ordnung ist zwar kein Hauptidealring, hat jedoch die Eigenschaft, dass jedes Ideal von maximal zwei Elementen erzeugt werden kann, von denen im Falle ganzer Ideale eines sogar in $\mathbb{Z}$ gewählt werden kann (vgl. [PZ89], Kapitel 6.3). Damit ist es möglich, ein ganzes Ideal unabhängig vom Körpergrad n durch eine ganze rationale Zahl und eine algebraische Zahl darzustellen. Diese Darstellung ist insbesondere für die Multiplikation sehr vorteilhaft, so erhält man z.B. eine $\mathbb{Z}$–Basis eines Produkts zweier Ideale, indem man die beiden Elemente in der Zwei–Element–Darstellung des

ersten Ideals jeweils mit allen n Elementen der $\mathbb{Z}$–Basis des zweiten Ideals multipliziert und aus den erhaltenen $2n$ Komponenten mittels Algorithmus 3.3.7 eine Basis konstruiert. Dadurch kann die Multiplikation wesentlich beschleunigt werden – sowohl im Vergleich zur Darstellung mittels einer $\mathbb{Z}$–Basis als auch im Vergleich zu unserer Darstellung. In der Praxis jedoch ist mit der Berechnung der Zwei–Element–Darstellung ein deutlicher Overhead verbunden, d.h. schon aus dem Ergebnis einer Multiplikation wieder einer Zwei–Element–Darstellung zu rekonstruieren ist aufwändig. Bei weniger aufwändigen Operationen, etwa der Addition, fällt der Gewinn, den man durch die Zwei–Element–Darstellung erreichen kann, noch deutlich geringer aus, das Problem, eine Zwei–Element–Darstellung für das Ergebnis zu finden, bleibt hingegen so aufwändig wie zuvor, so dass sich nach unserer Auffassung die Verwendung der Zwei–Element–Darstellung im allgemeinen nicht lohnt.

Ein wesentlicher Vorteil der Zwei-Element–Darstellung ist jedoch, dass sie eine Darstellung eines ganzen Ideals mit nur einer rationalen ganzen Zahl und einer algebraisch ganzen Zahl erlaubt, d.h. mit $n + 1$ rationalen ganzen Zahlen. Daher wählen wir diese Darstellung für Primideale, von denen wir eine große Menge in der Faktorbasis verwalten müssen, die unser Algorithmus zur Berechnung der Klassenzahl verwendet, so dass dem Speicherplatzbedarf eine kritische Bedeutung zukommt. Insbesondere zeigt sich hier auch ein zweiter Vorteil dieser Darstellung, die Zwei–Element–Darstellung erlaubt nämlich das Potenzieren von Primidealen durch einfaches Potenzieren der beiden Erzeuger. Da gerade das Potenzieren von Primidealen bei den Algorithmen zur Berechnung von Klassengruppen neben dem Aufmultiplizieren solcher Potenzen eine häufig ausgeführte Operation ist, enthält die Klasse `prime_ideal` nun die beiden Komponenten `gen1` und `gen2` vom Typ `bigint` bzw. `alg_number`, wobei `gen1` gerade die Primzahl beinhaltet, über der das Primideal liegt. Damit nutzen wir an den wesentlichen Stellen die Vorteile der Zwei–Element–Darstellung, ohne dafür mit den damit verbundenen Nachteilen bezahlen zu müssen. Um nun alle Funktionen, die Ideale als Argumente nehmen, auch für Primideale benutzen zu können, erhält die Klasse `prime_ideal` einen Cast–Operator, der ein Objekt vom Typ `prime_ideal` automatisch in ein Objekt vom Typ `alg_ideal` konvertiert, d.h. der Operator berechnet die in Kapitel 3.1 beschriebene Darstellung des Primideals.

Darüber hinaus enthält ein Objekt des Typs `prime_ideal` jeweils auch noch eine algebraische Zahl `valu`, die dazu benutzt wird, schnell zu berechnen, in

welcher Potenz dieses Primideal ein anderes Ideal teilt, sowie den Trägheitsgrad und den Verzweigungsgrad des Primideals. Daher sieht die Deklaration der Klasse `prime_ideal` wie folgt aus:

```
class prime_ideal
{
  friend class single_factor<alg_ideal>;

  bigint gen1;
  alg_number gen2;
  lidia_size_t e,f;        // ramifaction,inertia
  mutable alg_number valu; // A value helpful for
                           // computing valuations

  void compute_valu() const;

public:
  prime_ideal();
  prime_ideal(const bigint&, const alg_number&,
              lidia_size_t ram = 0, lidia_size_t inert = 0);
  prime_ideal(const bigint&,
              const nf_base * O = nf_base::current_base);
  ~prime_ideal() {};

  // Convert to ideal
  operator alg_ideal() const
  { return alg_ideal(gen1, gen2);}

  // Access Functions
  const bigint & first_generator() const;
  const alg_number & second_generator() const;
  lidia_size_t ramification_index() const;
  lidia_size_t degree_of_inertia() const;
  const bigint & base_prime() const;

  // swap
  friend void swap(prime_ideal &a, prime_ideal &b);
```

```
  // Higher Level Functions
  friend bigint norm(const prime_ideal & p);
  friend bigint exponent(const prime_ideal & p);
  friend long ord(const prime_ideal &, const alg_ideal &);
            // ord_p(m);
  friend long ord(const prime_ideal &, const alg_number &);
            // ord of principal ideal
  friend void power(alg_ideal &,
                    const prime_ideal &, const bigint &);

  // In-/Output:
  friend ostream& operator<<(ostream &, const prime_ideal &);
  friend istream& operator>>(istream &, prime_ideal &);
};
```

Speziell zu der Funktion `ord`, die berechnet, welches die maximale Potenz des Primideals ist, die das Ideal noch teilt, wollen wir anlässlich der Idealfaktorisierung an späterer Stelle mehr sagen.

4.2 Primzahlzerlegung

Da wir später Ideale als Potenzprodukt von Primidealen schreiben wollen, besteht eine wichtige Teilaufgabe darin, diese Primideale überhaupt zu bestimmen. Dies geschieht, indem wir das von einer Primzahl erzeugte Hauptideal in seine Primideale zerlegen. Ähnlich wie bei der Berechnung der Maximalordnung gibt es auch hier wieder zwei Algorithmen. Einen einfachen Algorithmus auf der Basis der Polynomarithmetik über endlichen Körpern, der jedoch nicht in allen Fällen ausreicht, und einen aufwändigeren Algorithmus, der sich Linearer Algebra über endlichen Körpern bedient. Der einfachere Algorithmus beruht auf dem folgenden Resultat:

4.2.1. Satz

Sei $K = \mathbb{Q}[\rho]$ ein algebraischer Zahlkörper, wobei ρ algebraisch ganz sei und das Minimalpolynom $A \in \mathbb{Z}[X]$ habe. Sei p eine Primzahl, die nicht den Index teilt, d.h. $p \nmid [\mathcal{O}_K : \mathbb{Z}[\rho]]$, bezeichne $\bar{\cdot}$ Reduktion modulo p, zerfalle $\overline{A}(X)$ in $\mathbb{F}_p[X]$

in irreduzible Faktoren:

$$\overline{A}(X) = \prod_{i=1}^{k} \overline{A_i}(X)^{e_i}$$

in $\mathbb{F}_p[X]$*, und sei* $A_i \in \mathbb{Z}[X]$ *ein beliebiges normiertes Urbild von* $\overline{A_i}$ *(unter* $\bar{\cdot}$*). Dann gilt:*

$$p\mathcal{O}_K = \prod_{i=1}^{k} (p\mathcal{O}_K + A_i(\rho)\mathcal{O}_K)^{e_i}.$$

Dabei sind die $p\mathcal{O}_K + A_i(\rho)\mathcal{O}_K$ *Primideale, deren Restklassengrad* f_i *gleich dem Grad von* A_i *ist.*

Zum Beweis vergleiche [Coh95], Kapitel 4.8.2.

Im Falle, dass p den Index nicht teilt, ist die Idealfaktorisierung also mittels Polynomfaktorisierung in endlichen Primkörpern ausführbar. Hierfür gibt es schon seit langem effiziente Algorithmen. Für den Fall der Indexteiler benötigen wir allerdings das folgende kompliziertere Resultat:

4.2.2. Satz

Sei p *eine Primzahl und* $\mathcal{O}$ *eine Ordnung eines algebraischen Zahlkörpers* K *mit* $\mathrm{ggT}([\mathcal{O}_K : \mathcal{O}], p) = 1$ *und* I_p *bezeichne wieder das* p*–Radikal. Dann hat* $p\mathcal{O}$ *eine eindeutige Zerlegung in maximale Ideale in* $\mathcal{O}$*:*

$$p\mathcal{O} = \prod_{\mathfrak{m}|p} \mathfrak{m}^{e(\mathfrak{m})},$$

wobei die $\mathfrak{m}$*, die* $p\mathcal{O}$ *genau in* j*–ter Potenz teilen, gerade den Komponenten entsprechen, die man bei der Zerlegung der freien separablen* $\mathbb{F}_p$*–Algebra* $\mathcal{O}/C_j$ *mit*

$$C_j := ({I_p}^j + p\mathcal{O})^2((I_p + p\mathcal{O})^{j-1}(I_p + p\mathcal{O})^{j+1})^{-1}$$

in ein Produkt endlicher Körper erhält.

Die Primideale in der Maximalordnung $\mathcal{O}_K$ *ergeben sich gerade als* $\mathfrak{m} \cdot \mathcal{O}_K$*.*

Zum Beweis vergleiche [BLJ], Kapitel 6.

Eine ausführliche Beschreibung einer Implementierung des auf diesem Ergebnis basierenden Primzahlzerlegungsalgorithmus findet sich in [Web93]. Auf der Implementierung von Damian Weber basiert auch die von uns tatsächlich verwandte Primzahlzerlegungsroutine.

4.3 Idealfaktorisierung

Ein Kernpunkt der Klassengruppenberechnung ist stets die Zerlegung von Idealen in ein Produkt von Primidealen. Diese Faktorisierung ist mindestens so schwer zu berechnen, wie die ganzzahlige Faktorisierung der Norm des Ideals in Primzahlen, wie man daran sieht, dass man wegen der Multiplikativität der Norm aus einer Idealfaktorisierung die Faktorisierung der Norm leicht ablesen kann.

Daher scheint es zulässig, die Faktorisierung der Norm über den ganzen Zahlen als Teilroutine der Idealfaktorisierung zu verwenden. Tatsächlich geht es uns dabei allerdings – wie wir sehen werden – nur darum, die Primzahlen zu finden, die die Norm teilen. Daher reicht es auch, den Exponenten des Ideals zu faktorisieren.

Der Algorithmus zur Idealfaktorisierung funktioniert nun so: Bestimme den Exponenten des Ideals $\mathfrak{a}$ und berechne seine Primteiler (mit Hilfe eines Algorithmus zur Faktorisierung ganzer Zahlen). Zerlege alle diese Primteiler jeweils in ein Produkt von Primidealen wie im vorangehenden Kapitel beschrieben. Dies ergibt die Menge der Kandidaten, die das Ideal teilen können. Bestimme nun für jeden Kandidaten $\mathfrak{p}$, in welcher Potenz er das Ideal teilt. Diese Potenz bezeichnet man auch als $\mathrm{ord}_{\mathfrak{p}}(\mathfrak{a})$. Dies könnte etwa durch Probedivision geschehen. Da eine Idealdivision jedoch sehr teuer ist, suchen wir ein besseres Verfahren. Der folgende Satz zeigt, wie man die Division zweier Ideale im wesentlichen durch die Multiplikation eines Ideals mit einer algebraischen Zahl ersetzen kann:

4.3.1. Satz
Sei p eine Primzahl und $\mathcal{O}$ eine Ordnung eines algebraischen Zahlkörpers K mit $\mathrm{ggT}([\mathcal{O}_K : \mathcal{O}], p) = 1$ *und $\mathfrak{p}$ ein Primideal über p. Dann gibt es ein $a \in K \setminus \mathcal{O}$ mit $a\mathfrak{p} \subset \mathcal{O}$. Für ein beliebiges ganzes Ideal $\mathfrak{a} \subset \mathcal{O}$ gilt $\mathfrak{p} \mid \mathfrak{a}$ genau dann, wenn $a\mathfrak{a}$ ganz ist. Damit ist insbesondere* $\mathrm{ord}_{\mathfrak{p}}(\mathfrak{a})$ *genau die größte ganze Zahl v mit $a^v\mathfrak{a} \subset \mathcal{O}$.*

Der Beweis ergibt sich aus [Coh95], Proposition 4.8.15 und Lemma 4.8.16, wenn man berücksichtigt, dass unsere schwächere Voraussetzung bereits ausreicht, um die Invertierbarkeit von $\mathfrak{p}$ zu gewährleisten.

4.4 Implementierung der Idealfaktorisierung

Zum Zerlegen von Idealen in ein Potenzprodukt von Idealen (in unseren Anwendungen stets ein Potenzprodukt von Primidealen), sowie zur Manipulation solcher Zerlegungen, dienen zwei spezielle Klassen in LiDIA. Da es nicht nur für Ideale in algebraischen Zahlkörpern, sondern z.B. auch für ganze Zahlen und Polynome von Interesse ist, solche Zerlegungen zu handhaben, wurden diese Klassen als Templateklassen implementiert: `single_factor<T>` und `factorization<T>`. Dabei ist insbesondere die Klasse `factorization<T>` völlig generisch, während für `single_factor<T>` natürlich Spezialisierungen notwendig sind, etwa um Funktionen einzubinden, die ein solches Objekt in ein Potenzprodukt zerlegen und daher sinnvollerweise Namen haben, die vom Typ `T` abhängen. Funktionen, die für alle Typen gleich sind, also etwa die Multiplikation von zwei Objekten des Typs `single_factor<T>`, die einfach auf die Multiplikation von zwei Objekten des Typs `T` zurückgeführt wird, sind daher in der Klasse `base_factor<T>` definiert, von der die Spezialisierungen für `single_factor<T>` abgeleitet werden.

Dabei müssen natürlich Kompromisse geschlossen werden. So wäre es z.B. für unseren Fall wünschenswert, Primideale gleich als `prime_ideal` abzuspeichern, andererseits will man für ganze Zahlen und Polynome aber sicher nicht verschiedene Typen für die irreduziblen Faktoren und die reduziblen Faktoren verwenden. Ein solcher Kompromiss beruht zum Beispiel darauf, dass man für ein Ideal $\mathfrak{a}$ und ein Primideal $\mathfrak{p}$ sehr viel leichter $\mathrm{ord}_{\mathfrak{p}}(\mathfrak{a})$ berechnen kann als mit Probedivision. Daher enthält die allgemeine Klasse `base_factor<T>` die Funktion `ord_divide`, die für den obigen Spezialfall erst diese Ordnung ausrechnet, und in einer einzigen Division gleich die richtige Potenz von $\mathfrak{p}$ aus dem Ideal $\mathfrak{a}$ herausdividiert. Im allgemeinen benutzt diese Funktion allerdings Probedivision. Aufbauend auf dieser Spezialfunktion lässt sich dann etwa der Factor–Refinement–Algorithmus ([BDS93]) in der Klasse `factorization<T>` in voller Allgemeinheit implementieren, bietet jedoch dank der Verwendung dieser Funktion in unserer speziellen Situation die volle Effizienz. Hier zahlt sich auch für Polynome und ganze Zahlen die Aufteilung in Primfaktoren und (potenziell) zusammengesetzte Faktoren aus, da man nun systematisch vermeiden kann, Paare verschiedener Primfaktoren überhaupt anzusehen. Für eine detailliertere Beschreibung der Funktionalität der Klasse `factorization<T>` verweisen wir auf [98].

Trotz dieses Kompromisses erweist es sich jedoch als notwendig, die Klasse `single_factor<alg_ideal>` unabhängig von der Basisklasse `base_factor<T>` zu implementieren, da andernfalls auch jedes `prime_ideal` als `alg_ideal` repräsentiert werden müsste. Darunter würde jedoch in hohem Maße die Effizienz leiden, da ständig aufwändige Konvertierungen von Objekten der Klasse `prime_ideal` in Objekte der Klasse `alg_ideal` notwendig wären und bei jedem Zugriff auf einen Primfaktor in einer Zerlegung erst wieder ein Objekt der Klasse `prime_ideal` konstruiert werden müsste.

Daher enthält ein Objekt der Klasse `single_factor<alg_ideal>` wahlweise ein Objekt der Klasse `prime_ideal` oder ein Objekt der Klasse `alg_ideal`. Dadurch wird in allen Routinen eine Fallunterscheidung notwendig, so dass die Klasse nun unabhängig von der allgemeinen Basisklasse `base_factor<T>` implementiert werden muss. Immerhin können wir die Klasse aber noch von der Klasse `decomposable_object` ableiten, die für alle Faktorisierungen und die einzelnen Faktoren die Informationen über Primalität handhabt.

Daher ergibt sich die folgende Klassendefinition:

```
class single_factor<alg_ideal>: public decomposable_object
{
  friend class factorization<alg_ideal>;

private:
  // Use one of the following two variables.
  // Which one depends on prime_flag().
  prime_ideal rep_p;
  alg_ideal rep_a;

  static bool verbose_flag;

public:
  inline static bool verbose();
  inline static void set_verbose_mode(bool b);

  /**
   ** constructors, destructor
   **/
```

```
single_factor();
// default value must be '1' (neutral element)
single_factor(const single_factor<alg_ideal> &);
//copy constructor
single_factor(const alg_ideal&);
//conversion for type alg_ideal
single_factor(const prime_ideal &);
//conversion for type prime_ideal
~single_factor(){}
//destructor

/**
 ** queries
 **/

bool is_prime_factor() const;

bool is_prime_factor(int test);
// test==0 -> no explicit primality test (check flag)
// test!=0 -> explicit prime-test if prime_state()==unknown
//see manual

alg_ideal extract_unit();  // always return the order

bool is_one() const;

/**
 ** access function
 **/

const alg_ideal& base() const;
alg_ideal& base ();

const prime_ideal&  prime_base() const;
prime_ideal&  prime_base();
```

```
  /**
   ** swap
   **/

public:
  friend void swap(single_factor<alg_ideal> & a,
                   single_factor<alg_ideal> & b);

  /**
   ** assignment
   **/

  single_factor<alg_ideal> &
  operator =(const single_factor<alg_ideal> & x);
  const alg_ideal & operator =(const alg_ideal & x);
  const prime_ideal & operator =(const prime_ideal & x);
  void assign(const single_factor<alg_ideal> & x);
  void assign(const alg_ideal & x);
  void assign(const prime_ideal & x);

  /**
   ** arithmetic operations
   **/

  friend void multiply(single_factor<alg_ideal> & c,
                       const single_factor<alg_ideal> & a,
                       const single_factor<alg_ideal> & b);

  friend void divide(single_factor<alg_ideal> & c,
                     const single_factor<alg_ideal> & a,
                     const single_factor<alg_ideal> & b);
  // ********************************************************
  // *WARNING : we cannot check if 'b' is a divisor of 'a' ! *
  // ********************************************************

  friend lidia_size_t
```

```
ord_divide(const single_factor<alg_ideal> &a,
           single_factor<alg_ideal> &b);

/**
 ** factorization algorithms
 **/

factorization<alg_ideal> factor(int upper_bound = 34) const;
// standard factorization algorithm, used in function
// factorization<alg_ideal>::factor_all_components
void factor(factorization<alg_ideal> &,
            int upper_bound = 34) const;
friend void factor(factorization<alg_ideal> &,
                   const alg_ideal &, int upper_bound = 34);
friend factorization<alg_ideal> factor(const alg_ideal &,
                                       int upper_bound = 34);

friend void decompose_prime(prime_ideal * &factor,
                            lidia_size_t & num,
                            const bigint & p, const order & O);

factorization<alg_ideal> finish(rational_factorization&) const;
friend factorization<alg_ideal> finish(const alg_ideal &,
                                       rational_factorization &);
void finish(factorization<alg_ideal> &,
            rational_factorization &) const;
friend void finish(factorization<alg_ideal> &,
                   const alg_ideal &, rational_factorization &);

factorization<alg_ideal> trialdiv(
                             unsigned int upper_bound=1000000,
                             unsigned int lower_bound=1) const;
void trialdiv(factorization<alg_ideal> &,
              unsigned int upper_bound=1000000,
              unsigned int lower_bound=1) const;
friend factorization<alg_ideal> trialdiv(const alg_ideal &,
```

```
                                unsigned int upper_bound=1000000,
                                unsigned int lower_bound = 1);
  friend void trialdiv(factorization<alg_ideal> &,
                       const alg_ideal &,
                       unsigned int upper_bound=1000000,
                       unsigned int lower_bound = 1);

  factorization<alg_ideal> ecm(int upper_bound = 34,
                               int lower_bound = 6,
                               int step = 3) const;
  friend factorization<alg_ideal> ecm(const alg_ideal &,
                                      int upper_bound = 34,
                                      int lower_bound = 6,
                                      int step = 3);
  void ecm(factorization<alg_ideal> &, int upper_bound = 34,
           int lower_bound = 6, int step = 3) const;
  friend void ecm(factorization<alg_ideal> &,
                  const alg_ideal &, int upper_bound = 34,
                  int lower_bound = 6, int step = 3);

  factorization<alg_ideal> mpqs() const;
  friend factorization<alg_ideal> mpqs(const alg_ideal &);
  void mpqs(factorization<alg_ideal> &) const;
  friend void mpqs(factorization<alg_ideal>&, const alg_ideal &);

  /**
   ** misc
   **/

//we need new comparisons because of the flag 'know_modulus'
  friend bool operator ==(const single_factor<alg_ideal> & a,
                          const single_factor<alg_ideal> & b);
  friend bool operator !=(const single_factor<alg_ideal> & a,
                          const single_factor<alg_ideal> & b);

  friend void gcd(single_factor<alg_ideal>&c,
```

```
                const single_factor<alg_ideal>&a,
                const single_factor<alg_ideal>&b);
  friend single_factor<alg_ideal>
  operator /(const single_factor<alg_ideal>&a,
             const single_factor<alg_ideal>&b);
  //
  // a specialization of the output routine
  // which produces nicer output.
  //
  friend ostream & operator <<(ostream &out,
                               const single_factor<alg_ideal> &f);
  friend istream & operator >> (istream &in,
                                single_factor<alg_ideal> &f);
};

// The routine for factoring an index divisor p
// in a p-maximal order of a number field

void factor_p(const bigint & prime, const order & Order,
              lidia_size_t &number, prime_ideal* & ideals);

void gcd(single_factor<alg_ideal>&c,
         const single_factor<alg_ideal>&a,
         const single_factor<alg_ideal>&b);

bool operator !=(const single_factor<alg_ideal> & a,
                 const single_factor<alg_ideal> & b);

single_factor<alg_ideal>
operator *(const single_factor<alg_ideal>&a,
           const single_factor<alg_ideal>&b);
single_factor<alg_ideal>
operator /(const single_factor<alg_ideal>&a,
           const single_factor<alg_ideal>&b);

ostream & operator <<(ostream &out,
```

```
                        const single_factor<alg_ideal> &f);
istream & operator >> (istream &in,
                       single_factor<alg_ideal> &f);

inline bool operator < (const single_factor<alg_ideal> & a,
                        const single_factor<alg_ideal> & b);
inline bool operator > (const single_factor<alg_ideal> & a,
                        const single_factor<alg_ideal> & b);
inline bool operator <= (const single_factor<alg_ideal> & a,
                         const single_factor<alg_ideal> & b);
inline bool operator >= (const single_factor<alg_ideal> & a,
                         const single_factor<alg_ideal> & b);
```

Erklärungsbedürftig sind hier die Funktionen zur eigentlichen Faktorisierung. Wie in Kapitel 4.3 beschrieben, faktorisieren wir hierzu zunächst den Exponenten des Ideals um dann mittels der Funktion `decompose_prime` die Zerlegung aller auftretenden Primzahlen in ihre Primideale zu bestimmen. Zum Schluss bestimmen wir dann mittels `ord_divide`, in welcher Potenz jedes der gefundenen Primideale das zu faktorisierende Ideal teilt. Dabei ist der dominierende Faktor in der Laufzeit die Faktorisierung des Exponenten, sofern der Exponent nicht besonders leicht zu faktorisieren ist. Daher stellen wir ein breites Spektrum von Faktorisierungsmethoden zur Verfügung, die sich allerdings alle nur in der Methode unterscheiden, die zur Faktorisierung des Exponenten benutzt wird. Wir duplizieren dabei das Interface der Klasse `rational_factorization`, d.h. die Erklärung der Funktionsnamen und ihrer Parameter ergibt sich aus der Beschreibung der Klasse `rational_factorization`. Eine natürlichere Methode, um diese Möglichkeit anzubieten, hätte darin bestanden, *eine* Faktorisierungsfunktion anzubieten und dieser als Parameter eine Funktion zur Faktorisierung des Exponenten und eventuelle Parameter für diese Funktion zu übergeben, dem steht allerdings die breite Varianz in den Argumenttypen für die Funktionen zur Faktorisierung ganzer Zahlen entgegen, so dass wir mittels variabler Argumentlisten die Typprüfung für Aufrufe dieser Faktorisierungsfunktion weitgehend hätten außer Kraft setzen müssen. Zugunsten einer genaueren Typprüfung bei der Übersetzung der C++–Programme haben wir diese Variante daher verworfen.

Kapitel 5

Klassengruppenberechnung

Nun wenden wir uns den eigentlichen Routinen zur Klassengruppenberechnung zu. Dazu erklären wir zunächst die allgemeine Idee der Klassengruppenberechnung mittels Relationensuche und HNF–Berechnung und wenden uns dann den Methoden der Relationensuche zu.

5.1 Klassengruppen– und Regulatorberechnung

Insbesondere für quadratische Zahlkörper gibt es eine Reihe von Verfahren zur Berechnung der Klassenzahl. Diese basieren etwa auf der Theorie quadratischer Formen oder für imaginärquadratische Zahlkörper, wo bekannt ist, dass der Regulator stets 1 ist, auch auf der analytischen Klassenzahlformel (vgl. etwa [Coh95], Kapitel 5 oder [BW97]). Diese liefern jedoch keine Aussage über die Struktur der Klassengruppe. Will man diese auch berechnen, so gibt es zur Zeit zwei Methoden.

Zunächst bietet sich Shanks' „Baby Step – Giant Step" – Algorithmus an, der in jeder Gruppe funktioniert, jedoch eine Laufzeit hat, die exponentiell in der Größe der Gruppe ist. Außerdem ist es dabei schwierig, den Regulator zu finden.

Eine asymptotisch bessere Laufzeit hat ein Algorithmus, der auf einer Idee von McCurley beruht und von Buchmann verallgemeinert wurde ([Buc89]). Auf der Grundidee dieses Algorithmus beruht unser Verfahren, so dass wir ihn zunächst darstellen wollen.

Sei dazu K ein algebraischer Zahlkörper vom Grad n mit r reellen und $2s$ komplexen Einbettungen (also $n = r+2s$) und FB eine Menge von Primidealen, die die Klassengruppe $Cl(K)$ erzeugen und sei $k = |FB|$, so dass wir schreiben

können: $FB = \{\mathfrak{p}_1, \ldots, \mathfrak{p}_k\}$. Solch eine fixierte Menge von Primidealen bezeichnen wir im weiteren als *Faktorbasis*.

Mit A_{FB} wollen wir die Menge aller algebraischen Zahlen α bezeichnen, die die Eigenschaft haben, dass das von ihnen erzeugte Hauptideal sich als Potenzprodukt der Ideale in FB darstellen lässt, d.h.

$$A_{FB} = \left\{ \alpha \in K | \text{es gibt } e_1, \ldots, e_k \in \mathbb{Z} \text{ mit } \alpha \cdot O_K = \prod_{i=1}^{k} \mathfrak{p}_i^{e_i} \right\}.$$

Wir betrachten die Abbildungen

$$\begin{aligned} \Phi' : A_{FB} &\longrightarrow \mathbb{Z}^k \\ \alpha &\longmapsto (e_1, \ldots, e_k) \end{aligned}$$

und

$$\begin{aligned} \Phi : A_{FB} &\longrightarrow \mathbb{Z}^k \times \mathbb{R}^{r+s-1} \\ \alpha &\longmapsto (e_1, \ldots, e_k, \log|\sigma_1(\alpha)|, \ldots, \log|\sigma_{r+s-1}(\alpha)|), \end{aligned}$$

wobei die e_i die Koeffizienten in der Zerlegung $\alpha \cdot O_K = \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}$ sind.

Die Tupel $(e_1, \ldots, e_k)$ aus dem Bild von Φ' nennen wir im weiteren Verlauf *Relationen* und die Tupel $(e_1, \ldots, e_k, \log|\sigma_1(\alpha)|, \ldots, \log|\sigma_{r+s-1}(\alpha)|)$ *erweiterte Relationen*.

Laut [Buc89] (Theorem 2.1) gilt für das Gitter der (erweiterten) Relationen:

5.1.1. Satz
Unter den gegebenen Voraussetzungen ist $\Phi(A_{FB})$ ein $(k + r + s - 1)$–dimensionales Gitter mit Determinante hR.

$\Phi'(A_{FB})$ ist ein k–dimensionales Gitter mit Determinante h und es ist $\mathbb{Z}^k/\Phi'(A_{FB}) \cong Cl(K)$.

Nun müssen wir noch eine Bedingung finden, die uns garantiert, dass eine Menge FB in der Tat die Klassengruppe erzeugt. Dazu berufen wir uns auf eine Arbeit von E. Bach ([Bac90]) aus der sich unter Voraussetzung der verallgemeinerten Riemannschen Vermutung das folgende Resultat ergibt:

5.1.2. Satz
Sei K ein Zahlkörper mit Diskriminante Δ. Dann erzeugen die Primideale primer Norm mit Norm $\leq 18(\log \Delta)^2$ die Klassengruppe. Lassen wir alle Primideale zu, so erzeugen bereits diejenigen mit Norm $\leq 12(\log \Delta)^2$ die Klassengruppe.

Darüber hinaus lassen sich in Abhängigkeit von der Anzahl der reellen bzw. komplexen Einbettungen des Körpers bessere Schranken angeben, siehe [Bac90]. Daneben gibt es auch Abschätzungen, die unabhängig von der Riemannschen Vermutung sind. So erzeugen in einem Zahlkörper K mit Körpergrad n, Diskriminante Δ und $2t$ komplexen Einbettungen z.B. auch die Primideale mit Norm

$$\leq \frac{n!}{n^n} \left(\frac{4}{\pi}\right)^t \sqrt{|\Delta|} =: M_K$$

die Klassengruppe (siehe [PZ89], Kapitel 6.2). M_K heißt dabei Minkowski–Konstante von K. Obwohl diese Schranke asymptotisch natürlich wesentlich schlechter ist, als die zuvor gegebene Abschätzung, ist sie aufgrund der wesentlich kleineren Konstanten im Bereich kleiner Diskriminanten dennoch besser, beispielsweise im Falle total–reeller Zahlkörper vom Grad 10 für bis zu 15–stellige Diskriminanten.

Damit ist das Problem der Klassengruppenberechnung darauf reduziert, Relationen über einer geeigneten Faktorbasis zu bestimmen und anschließend die Determinante von $\Phi(A_{FB})$ und die Hermite–Normalform von $\Phi'(A_{FB})$ zu bestimmen. Diesen Problemen wollen wir uns nun zuwenden, insbesondere wollen wir im weiteren Verlauf sehen, wie man zur Relationensuche das Number Field Sieve einsetzten kann.

Dabei ist ein besonderes Problem die Frage nach der Darstellung der Relationen. Wenn wir wirklich Logarithmenvektoren $(\log|\sigma_1(\alpha)|, \ldots, \log|\sigma_{r+s-1}(\alpha)|)$ abspeichern, entstehen im Laufe der Hermite–Normalform–Berechnung Präzisionsprobleme. Speichern wir statt der Logarithmen jeweils die algebraische Zahl α selbst ab, entstehen im Laufe der Berechnung sehr große Einträge. Eine gut geeignete Darstellung ist wahrscheinlich die sogenannte *kurze Darstellung*, wie sie in [Thi95] beschrieben wird.

5.2 Relationengenerierung mittels Reduktionstheorie

Zunächst wollen wir jedoch die ursprüngliche Idee zur Relationengenerierung vorstellen, die bei der Vorstellung des Algorithmus vorgeschlagen wurde, da wir diese Methode später ebenfalls benötigen werden. Sie beruht auf der sogenannten Reduktionstheorie. Zu einem gegebenen ganzen Ideal $\mathfrak{a}$ berechnen wir dabei ein äquivalentes ganzes Ideal $\mathfrak{a}'$ indem wir in $\mathfrak{a}$ ein *Minimum* im folgenden Sinne

bestimmen:

5.2.1. Definition
Sei $\mathfrak{a}$ *ein (ganzes) Ideal eines algebraischen Zahlkörpers* K. $\alpha \in \mathfrak{a}$ *heißt* Minimum von $\mathfrak{a}$, *falls für alle* $\beta \in \mathfrak{a}$ *gilt:*

$$(\forall i |\sigma_i(\beta)| < |\sigma_i(\alpha)|) \Longrightarrow \beta = 0.$$

Solch ein Minimum α hat die Eigenschaft, dass $\mathfrak{a}/\alpha$ wieder ein ganzes Ideal ist ([Buc89]).

In der Praxis ist es schwierig ein solches Minimum zu berechnen. Daher gehen wir statt dessen so vor, dass wir eine Darstellung mittels Konjugiertenvektoren berechnen und dann mittels LLL–Reduktion eine Zahl α in dem Ideal bestimmen, so dass α einen kurzen Konjugiertenvektor hat. Dann ist i.a. zwar $\mathfrak{a}/\alpha$ kein ganzes Ideal mehr, allerdings genügt meistens eine kleine natürliche Zahl n, um mit $\mathfrak{a}/\frac{\alpha}{n}$ doch ein ganzes Ideal zu erhalten.

Insgesamt ergibt sich damit folgendes Verfahren, um eine Relation zu finden:

Berechne ein zufälliges Potenzprodukt $\mathfrak{a}$ der Primideale in der Faktorbasis und *reduziere* dieses mit einer geeigneten algebraischen Zahl α, so dass $\mathfrak{a}' := \mathfrak{a}/\alpha$ wieder ein ganzes Ideal ist. Dann versuche $\mathfrak{a}'$ über der Faktorbasis zu zerlegen. Gelingt dies, so hat man mit $\alpha = \mathfrak{a}/\mathfrak{a}'$ eine Relation gefunden.

In der Praxis zeigt sich dabei, dass es – wenn man zur Reduktion den LLL–Algorithmus verwendet – sowohl für den Zeitbedarf, als auch für die Qualität des Reduktionsergebnisses vorteilhaft ist, nicht zunächst das Potenzprodukt zu berechnen und dann die Reduktion auszuführen, sondern diese beiden Schritte zu kombinieren, indem bei der Berechnung des Potenzprodukts nach jeder Multiplikation oder Quadrierung gleich eine Reduktion angeschlossen wird.

5.3 Hilfsstrukturen und Funktionen zur Klassenzahlberechnung

Zur Berechnung der Klassenzahl brauchen wir also weitere Daten, die in keine der bisherigen Klassen passen. So müssen wir zum Beispiel eine Faktorbasis und Relationen speichern. Da hier unter Umständen ein sehr hoher Speicherplatzbedarf besteht, haben wir die benötigten Datenstrukturen in eine eigene Klasse gepackt. Auf diese Weise kann der Anwender flexibler handhaben, wo er diese Daten benutzen möchte, als wenn wir sie mit in die Klasse `order` gepackt hätten. Diese Klasse heißt `cgr_sieve` und hat das folgende Aussehen:

```
class cgr_sieve {
public:
  order O;

  // Relation matrix and factorbase.
  matrix<bigint> relations;
  sl_list<prime_ideal> factorbase;
  base_vector <alg_number> generators;

  // Information needed for simplifying relations:
  matrix<bigint> removing_relations;
  sl_list<prime_ideal> orig_base;
  base_vector <alg_number> orig_generators;

  lidia_size_t poly_limit;
  int pairs_per_relation;

  // Internal variables for converting the sieving results.
  int * a_act, * b_act, * index, * entry;
  int bach;
public:
  lidia_size_t rel_gen(lidia_size_t missing_ideal,
                       lidia_size_t column, lidia_size_t min_to,
                       lidia_size_t max_to);
  lidia_size_t rel_gen_reduction (lidia_size_t missing_ideal,
                                  lidia_size_t column,
                                  lidia_size_t min_to,
                                  lidia_size_t max_to);
  void sto_relation(base_vector<bigint> & relation,
                    lidia_size_t column);
  bool trialdiv(const alg_ideal &ideal1,
                base_vector<bigint> &exp_vector);
  void simplify();
  lidia_size_t sieve (const biv_hom_polynomial<bigint> &f,
                      int type,
                      const math_vector<bigint> & first_gen,
```

```
                    const math_vector<bigint> & second_gen,
                    const alg_ideal & ideal_to_divide_by,
                    const math_vector<bigint> & exp_of_ideal,
                    lidia_size_t first_column,
                    lidia_size_t min_to, lidia_size_t max_to);
  bigint cgr(matrix<bigint> & truerel,
             const order & Oinput, int bound);
  bigint cgr_finish(matrix<bigint> & truerel,
                    lidia_size_t * missing);
  bigint cgr_improve(matrix<bigint> & truerel,
                 long factor_limit);
  lidia_size_t * no_of_needed_relations(lidia_size_t total_num,
                                        const bigint_matrix & A);
};
```

Die Klasse enthält also neben einer Kopie der Ordnung, in der wir rechnen, insbesondere die Faktorbasis `factorbase`, die als einfach verkette Liste gespeichert wird (wozu wir die Templateklasse `sl_list` implementiert haben), eine Matrix, in der wir Relationen abspeichern (`relations`), und einen Vektor algebraischer Zahlen `generators`, in dem wir die Erzeuger der Hauptideale abspeichern, deren Zerlegung in der Relationenmatrix steht. Der i-te Eintrag des Vektors `generators` erzeugt also das Hauptideal, das wir auch als Potenzprodukt der Faktorbasis mit den Einträgen der i-ten Spalte der Matrix `relations` als Exponenten erhalten.

Mit Rücksicht auf weitere mögliche Anwendungen enthält die Klasse darüber hinaus in `removing_relations`, `orig_base` und `orig_generators` eine zweite Instanz dieser Informationen: Bei der Berechnung der Klassengruppe sind wir nämlich nur an den Primidealen interessiert, die einen nichttrivialen Beitrag zur Klassengruppe leisten und an den Relationen zwischen diesen Primidealen. Daher streichen wir aus `factorbase` alle Primideale, von denen wir erkennen, dass sie keinen Beitrag leisten und vereinfachen `relations` und `generators` dann jeweils entsprechend. Mit Blick auf die Berechnung des Regulators oder auf die Berechnung diskreter Logarithmen in der Klassengruppe fällt aber auf, dass auch dort wieder Relationen gefunden werden müssen, was über einer deutlich verkleinerten Faktorbasis allerdings schwierig wird. Daher nimmt die zweite Instanz die Informationen auf, die zur Streichung eines Primideals aus der Faktorbasis dienen, so dass wir später Relationen über der vollen

Faktorbasis suchen können, und diese mittels der gespeicherten Informationen leicht in eine Relation über der verkleinerten Faktorbasis umrechnen können. Diese Umrechnung findet in unserer Implementierung automatisch in der Funktion `sto_relation` statt.

Die Variablen `poly_limit` und `pairs_per_relation` werden intern benutzt, um abzuschätzen, ob es für ein gegebenes Polynom noch Sinn macht, zu versuchen, mit dem Siebtool Relationen zu finden und wenn ja, wie groß das Siebintervall in etwa gewählt werden muß.

Die Variablen `a_act`, `b_act`, `index` und `entry` dienen dazu, die Resultate des Siebtools aufzunehmen, dass wir zum Finden von Relationen benutzen, `bach` enthält die die verwendete Schranke, unterhalb der die Normen der verwendeten Primideale liegen müssen, z.B. die Bachschranke.

Die zentrale Funktion dieser Klasse ist `cgr`, die als Eingabe eine Ordnung und eine Schranke für die zu untersuchenden Primideale erhält und die Klassenzahl und eine Beschreibung der Struktur der Klassengruppe (in der Matrix `truerel`) zurückgibt.

Auf die weiteren Funktionen werden wir an den jeweiligen Stellen der weiteren Beschreibung eingehen.

5.4 Relationengenerierung mittels Siebverfahren

Nun wenden wir uns der Frage zu, wie man (erweiterte) Relationen finden kann. Dazu fixieren wir in diesem Kapitel nun einen Zahlkörper $K = \mathbb{Q}(\rho)$ vom Grad n, wobei ρ Nullstelle des Polynoms f ist. Alle Berechnungen finden dann entweder in der Maximalordnung $\mathcal{O}_K$ des Körpers K oder in der Gleichungsordnung $\mathbb{Z}[\rho]$ statt. Mit $\omega_1, \ldots, \omega_n$ wollen wir eine fixierte Basis von $\mathcal{O}_K$ bezeichnen. Zusätzlich fixieren wir eine Faktorbasis, d.h. eine Menge von Primidealen, über der die im Laufe des Verfahrens generierten Ideale zerlegt werden sollen.

Nun wollen wir zeigen, wie man in dieser Situation mittels der Ideen des Number–Field–Sieves Relationen finden kann, wozu wir zunächst die Grundidee des Number–Field–Sieves (in leicht verallgemeinerter Form) darstellen, und dann darauf eingehen, welche speziellen Probleme sich in unserer Situation ergeben und wie man sie lösen kann.

5.5 Die Grundidee

Das in Kapitel 5.2 beschriebene Verfahren zum Finden von (erweiterten) Relationen leidet darunter, dass man Ideale durch Probedivision in Produkte von Primidealen zerlegen muss. Selbst unter Verwendung sehr guter Heuristiken wird man in den meisten Fällen sehr viel Zeit in eine solche Probedivision investieren, und nur herausfinden, dass das Ideal nicht zerfällt. Dieses Problem ist vom Faktorisieren ganzer Zahlen her bekannt. Dort hat man dieses Problem gelöst, indem man Siebmethoden entwickelt hat, durch die es mit vergleichsweise geringem Aufwand möglich ist, die meisten Zahlen, die nicht zerfallen, a priori auszusortieren. Diese Methode und die dabei gewonnen Erfahrungen wollen wir erstmals auf das Problem der Klassengruppenberechnung in seiner allgemeinen Form anwenden.

Eine spezielle Anwendung der Ideen des quadratischen Siebs auf den Fall der quadratischen Zahlkörper zeigt Jacobson in [JJ98]. Diese Übertragung ist sehr erfolgreich, jedoch ist das quadratische Sieb für den allgemeinen Fall nicht anwendbar. Während das quadratische Sieb stets Polynome vom Grad 2 benutzt, zeichnet sich der asymptotisch beste bekannte Faktorisierungsalgorithmus, das sogenannte Number–Field–Sieve (NFS), jedoch dadurch aus, dass auch Polynome höheren Grades behandelt werden können. Im Verlaufe der Berechnung werden zudem sehr effizient zahlreiche Ideale von allerdings spezieller Gestalt aus einem fixierten algebraischen Zahlkörper (fast) in Primideale zerlegt. Daher bietet es sich an, diesen Teil des Number–Field–Sieves anzupassen und statt des quadratischen Siebs zu verwenden. Die Grundidee dieses (Teil–)Algorithmus wollen wir nun vorstellen. Eine umfassende Darstellung des Number–Field–Sieves findet sich in [BLP93]. Die Implementierung, die von uns benutzt wird, baut auf der in [Zay95] und [Web97] beschriebenen Implementierung auf, die viele Optimierungsmöglichkeiten (z.B. Sieben durch Aufaddieren von Logarithmen statt durch Division) implementiert, auf die wir hier nicht weiter eingehen wollen.

Ein wesentliches und leicht zu überprüfendes Kriterium, ob ein Ideal über einer fixierten Faktorbasis aus Primidealen zerfällt, ist das Zerfallsverhalten der Norm des Ideals. Zerfällt diese etwa über den Primzahlen, über denen die Primideale der Faktorbasis liegen, so zerfällt auch das Ideal selbst, falls *alle* Primideale über jeder dieser Primzahlen in der Faktorbasis enthalten sind. Falls nicht alle Primideale enthalten sind, so liefert dieses Kriterium zumindest

die Möglichkeit, schon zahlreiche Ideale auszufiltern, die mit Sicherheit nicht zerfallen. Um Hauptideale zu finden, die über unserer Faktorbasis zerfallen, betrachten wir daher zunächst die Normform $\mathcal{N}(x_1\omega_1 + \cdots + x_n\omega_n)$.

Die Kernidee, die in unserem Zusammenhang wesentlich ist, beruht darauf, dass für ein Polynom $f(x_1, \ldots, x_n)$ jede Primzahl $p \in \mathbb{P}$ die folgende Eigenschaft hat:

$$p \mid f(x'_1, \ldots, x'_n) \Longrightarrow p \mid f(x'_1 + l_1 p, \ldots, x'_n + l_n p) \; \forall\, l_1, \ldots, l_n \in \mathbb{Z}.$$

Um dies in einem Siebverfahren auszunutzen, geht man wie folgt vor: Man fixiert einen Menge T, typischerweise einen (Hyper-)Quader. Dann konstruiert man alle Tupel $(x'_1, \ldots, x'_n)$ mit modulo p verschiedenen Koordinaten, für die $f(x'_1, \ldots, x'_n)$ von p geteilt wird, und markiert alle Tupel als durch p teilbar, deren Koordinaten sich nur um Vielfache von p von einem solchen Tupel unterscheiden. Damit sich der Aufwand für die anfängliche Konstruktion solcher Tupel rechnet, muss man sehr große Mengen T betrachten. Dadurch entsteht aber auch ein sehr hoher Speicherbedarf. In der Praxis des Number–Field–Sieves hat es sich daher bewährt, die Variablen $x_3, \ldots, x_n$ jeweils auf 0 zu fixieren und nur x_1 und x_2 zu variieren. Aus Gründen des Speicherbedarfs berechnet man dann jeweils für ein fixiertes x_2 die modulo p verschiedenen Stellen, an denen p $f(x_1, x_2, 0, \ldots, 0)$ teilt, und „siebt“ dann in Richtung von x_1, d.h. man markiert alle Stellen, deren erste Koordinate sich nur um ein Vielfaches von p von einer dieser Stellen unterscheidet. Diese Berechnung führt man dann für verschiedene Werte von x_2 aus.

Im Number–Field–Sieve betrachtet man dabei das Polynom $f(x, y) := \mathcal{N}(x + y\rho)$, dessen Koeffizienten (bis auf Vorzeichen) gerade mit den Koeffizienten des Körperpolynoms übereinstimmen. In unserer Situation betrachten wir allgemeiner Polynome $f_{\alpha,\beta}(x, y) := \mathcal{N}(x\alpha + y\beta)$ für algebraisch ganze Zahlen α und β, die über $\mathbb{Q}$ linear unabhängig sein müssen. Im Gegensatz zum Number–Field–Sieve ist es damit auch möglich, dass nun Zahlen außerhalb der Gleichungsordnung auftreten. Wir fixieren ganze Zahlen $A_{\min}$, $A_{\max}$ und $B_{\max}$, sowie eine Schranke B für die Faktorbasis. Dann sieben wir mit allen Primzahlen aus $FB = \{p \in \mathbb{P} | p < B\}$ auf der Menge der Paare $P = \{(a, b) \in \mathbb{Z}^2 | A_{\min} \leq a \leq A_{\max}, 1 \leq b \leq B_{\max}\}$ und bestimmen Paare $(a, b) \in P$ mit

$$\mathrm{ggT}(a, b) \;=\; 1 \tag{5.1}$$

$$\mathcal{N}(a\alpha + b\beta) \quad = \quad \prod_{p \in FB} p^{f_p}, f_p \in \mathbb{N}_0. \tag{5.2}$$

Solch ein Paar und seine Zerlegung bezeichnet man als Relation. Dabei dient Bedingung (5.1) dazu, zu sichern, dass man nicht Relationen erhält, die sich nur durch einen kleinen ganzen Faktor unterscheiden, da diese offenbar keine neue Information beitragen können.

Um diese Paare effizient zu finden, benutzt man das Siebverfahren. Im Detail zerlegt man dabei zunächst einen festen Wert b jeweils erst in ein Produkt von Primzahlen, d.h. man berechnet eine Zerlegung $b = \prod_{i=1}^{k} q_i^{g_i}$. Dann gilt:

$$\text{ggT}(a, b) \neq 1 \Leftrightarrow a = lq_i \text{ für ein } l \in \mathbb{Z} \text{ und ein } i \in \{1, \ldots, k\}$$

und

$$p \mid \mathcal{N}(a\alpha + b\beta) \Longrightarrow p \mid \mathcal{N}((a + lp)\alpha + b\beta) \text{ für alle } l \in \mathbb{Z}.$$

Es genügt also, für fixiertes b jeweils alle Startwerte a (d.h. Werte mit $A_{\min} \leq a < A_{\min} + p$) zu berechnen, für die eine Primzahl p die Zahl $\mathcal{N}(a\alpha + b\beta)$ teilt, dann ergeben sich alle weiteren Stellen, an denen p teilt gerade durch die obige Beziehung. Probleme gibt es dabei nur, falls p gerade $\text{lc}(f)$ teilt, da es dann keinen modulo p eindeutig bestimmten Startwert gibt. In diesem Fall kann aber $f(x, y)$ nur durch p teilbar sein, falls $p \mid y$ und in diesem Fall muss *jeder* Wert für x geprüft werden.

Damit wird nun das folgende Verfahren zur Relationensuche ermöglicht:

5.5.1. Algorithmus

Siebalgorithmus

EINGABE: $f, b, A_{\min}, A_{\max}, FB'$
AUSGABE: Bitarray H_b mit

$$H_b(a) = \begin{cases} 1, & (a, b) \text{ erfüllt (5.1) und (5.2).} \\ 0, & \text{sonst.} \end{cases}$$

PHASE 1: GGT–SIEB

INITIALISIERUNG

(1) Berechne die Primfaktoren $q_1, \ldots, q_k$ von b.

```
(2)  for (a = A_min; a ≤ A_max; a++) do
(3)    H_b(a) = 1
(4)  od
     Siebphase
(5)  for (i = 1; i ≤ k; i++) do
(6)    Berechne minimales a ∈ [A_min, A_max] mit q_i | a
(7)    for (; a ≤ A_max; a = a + q_i) do
(8)      H_b(a) = 0
(9)    od
(10) od
     Phase 2: algebraisches Sieb
     Initialisierung
(11) for (a = A_min; a ≤ A_max; a++) do
(12)   if (H_b(a) == 1) then
(13)     S_b(a) = f(a, b)
(14)   fi
(15) od
     Siebphase
(16) for (each (p, z) ∈ FB') do
(17)   if (z ≠ ∞) then
(18)     Berechne minimales a ∈ [A_min, A_max] mit p | (a + bz)
(19)     for (; a ≤ A_max; a = a + p) do
(20)       if (H_b(a) == 1) then
(21)         while (p | S_b(a)) do
(22)           S_b(a) = S_b(a)/p
(23)         od
(24)       fi
(25)     od
(26)   else
(27)     if (p | b) then
(28)       for (a = A_min; a ≤ A_max; a++) do
(29)         if (H_b(a) == 1) then
```

```
(30)              while (p | S_b(a)) do
(31)                 S_b(a) = S_b(a)/p_i
(32)              od
(33)           fi
(34)        od
(35)     fi
(36)  fi
(37) od
     AUSWERTUNG
(38) for (a = A_min; a ≤ A_max; a++) do
(39)   if (S_b(a) ≠ 1) then
(40)     H_b(a) = 0
(41)   fi
(42) od
```

Durch naive Probedivision durch alle Primideale, die über einer der Primzahlen liegen, die an der Faktorisierung der Norm beteiligt waren, erhalten wir dann Relationen der Form

$$(a\alpha + b\beta) = \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}.$$

Effizienter können wir dabei folgendermaßen vorgehen. Wir berechnen zunächst alle Primideale $\mathfrak{p}_1, \ldots, \mathfrak{p}_l$, die über einer der Primzahlen aus der Faktorbasis liegen. Dann berechnen wir für jede Relation, die wir mit dem Siebverfahren gefunden haben, mit Satz 4.3.1 $\mathrm{ord}_{\mathfrak{p}_m}((a+b\rho)\mathcal{O}_K)$ für alle $\mathfrak{p}_m$, die über p_i liegen und erhalten damit die gewünschte Zerlegung. Diese Funktion ist in der Routine `trialdiv` implementiert. Die gesamte Siebfunktionalität einschließlich dieser Probedivision wird in unserer Implementierung durch die Funktion `sieve` zur Verfügung gestellt, wobei die zahlreichen Parameter dazu dienen, die Funktion möglichst flexibel an die im weiteren beschriebenen Anforderungen anpassen zu können. die Parameter `first_column`, `min_to` und `max_to` beschreiben, welcher Teil der Relationenmatrix mindestens bzw. höchstens gefüllt werden soll, in `f`, `first_gen` und `second_gen` werden das Polynom und die Erzeuger α und β übergeben.

Allerdings gibt es nun noch eine Reihe von neuen Problemen, die man etwa beim Faktorisieren mit dem NFS nicht hat:

- Der Körper, in dem das Sieben stattfindet, ist a priori festgelegt. Dadurch haben wir weniger Möglichkeiten, ein „gutes“ Polynom (d.h. eines mit kleinen Koeffizienten und möglichst vielen Nullstellen modulo kleiner Primzahlen) zu bestimmen, als beim Faktorisieren.

- Für jedes der Primideale, die die Klassengruppe erzeugen, muss mindestens eine (erweiterte) Relation, an der es beteiligt ist, gefunden werden. Darüber hinaus müssen wir sogar so viele (erweiterte) Relationen finden, dass das (erweiterte) Relationengitter vollen Zeilenrang hat.

- Der Schritt der Linearen Algebra, der nötig ist, um die Struktur der Klassengruppe zu bestimmen, findet in $\mathbb{Z}$ statt, was zu Problemen mit der Eintragsgröße führt. Im Gegensazu dazu wird beim Faktorisieren nur Lineare Algebra über $\mathbb{Z}/2\mathbb{Z}$ benötigt.

- Die Verifikation des Resultats ist aufwändiger.

Zur (teilweisen) Lösung der ersten beiden Probleme haben wir den zusätzlichen Freiheitsgrad erlaubt, dass wir statt $\mathcal{N}(a + b\rho)$ nun $\mathcal{N}(a\alpha + b\beta)$ mit zwei beliebigen, über $\mathbb{Q}$ linear unabhängigen algebraisch ganzen Zahlen α und β betrachten. Unklar ist allerdings, wie wir geeignete Zahlen α und β sowie das zugehörige Polynom $\mathcal{N}(a\alpha+b\beta)$ finden. Geeignete Heuristiken zur Bestimmung von α und β – insbesondere unter Berücksichtigung des Problems, dass wir zu jedem Primideal wenigstens eine Relation benötigen – wollen wir in den folgenden Abschnitten entwickeln. Dabei haben wir gegenüber dem NFS den Vorteil, verschiedene Polynome benutzen zu können, was wir auch tun werden.

Die Konstruktion des zugehörigen Polynoms, nachdem man zwei geeignete Zahlen α und β gefunden hat, ist hingegen einfach. Es bieten sich gleich mehrere Möglichkeiten. Zum Beispiel können wir aus den komplexen Einbettungen der beiden Zahlen durch Berechnen von Produktsummen die Koeffizienten konstruieren. Dabei hat man allerdings mit Rundungsfehlern zu kämpfen, daher haben wir diese Methode verworfen und statt dessen eine gewählt, die es uns gestattet, mit exakter Arithmetik zu rechnen: Wir bestimmen Normen für $b = 1$ und $n + 1$ verschiedenen Werte von a. Dann können wir das eindeutig bestimmte Interpolationspolynom vom Grad n bestimmen und müssen damit das gesuchte Polynom gefunden haben.

In den folgenden Abschnitten wollen wir darauf eingehen, wie wir die Polynome zum Sieben innerhalb der Ordnung bestimmen können, dann wollen wir

zeigen, wie man dies naiv übertragen kann, um in einem bestimmten Ideal zu sieben und wie man diese naive Idee mit Hilfe der Reduktionstheorie effizient machen kann. Schließlich wollen wir uns der Frage zuwenden, wie man zu all diesen Polynomen jeweils ein passendes Siebintervall finden kann.

5.6 Auswahl der Polynome

Nun wenden wir uns der Frage zu, wie die Zahlen α und β zu wählen sind, um ein geeignetes Polynom zu finden. Dazu beachten wir zunächst, dass die Wahrscheinlichkeit, dass eine zufällig gewählte Zahl über einer fixierten Faktorbasis aus kleinen Zahlen zerfällt, um so größer ist, je kleiner die Zahl ist, d.h. das Polynom sollte so gewählt sein, dass es möglichst kleine Werte erlaubt. Allerdings kennt im allgemeinen Fall niemand ein gutes Kriterium, wie die Koeffizienten so konstruiert werden können, dass möglichst kleine Werte des Polynoms gewährleistet sind, obwohl für quadratische Polynome seit der Analyse des quadratischen Siebs ein gutes Kriterium für die Koeffizienten bekannt ist. In Ermangelung eines besseren Kriteriums werden wir daher anstreben, möglichst *kleine* Koeffizienten zu erhalten. Allerdings ist der Zusammenhang zwischen Paaren algebraischer Zahlen (α, β) und den Koeffizienten von $\mathcal{N}(x\alpha + y\beta)$ wiederum so kompliziert, dass sich kein effektives Kriterium für die Auswahl von α und β ableiten lässt. Klar ist allerdings, dass die Koeffizienten Produktsummen der Konjugierten von α und β sind. Wenn man sich daher bemüht, α und β so zu wählen, dass die Konjugierten möglichst klein sind, werden die Koeffizienten des Polynoms $\mathcal{N}(x\alpha + y\beta)$ relativ klein sein.

Solche α und β können wir finden, indem wir die Konjugiertenvektoren einer Basis der Maximalordnung bestimmen und dann – etwa mittels Gitterbasisreduktionsalgorithmen – möglichst kurze linear unabhängige Vektoren in diesem Gitter bestimmen.

Dabei geben wir uns damit zufrieden, die in LiDIA implementierten Algorithmen zu benutzen, d.h. wir berechnen eine LLL–reduzierte Basis (vgl. [Wet98]). Evtl. könnte es sich lohnen, stärkere Reduktionsverfahren zu verwenden. Da diese jedoch aufwändig zu implementieren sind, haben wir darauf verzichtet, auszuprobieren, ob sich der höhere Berechnungsaufwand lohnt.

In der Praxis gehen wir dabei so vor, dass wir für eine LLL–reduzierte Basis $\{\omega_1, \ldots, \omega_n\}$ der Ordnung die Normformen $\mathcal{N}(x + y\omega_i), 1 \leq i \leq n$ mit $\omega_i \neq 1$ betrachten. Dabei stellen wir durch Sieben auf einem kleinen Testintervall fest,

wieviel Prozent der Relationen man von jeder dieser Formen in etwa erwarten kann. Dies haben wir in der Funktion `no_of_needed_relations` implementiert, der wir übergeben, wieviele Relationen wir insgesamt benötigen und durch welche Transformation die Zahlen $\omega_i, 1 \leq i \leq n$ aus der gegebenen Basis der Ordnung hervorgehen. Als Ergebnis liefert die Funktion ein Array zurück, dessen i-ter Eintrag angibt, wieviele Relationen unter Verwendung von ω_i gefunden werden sollen. Danach sieben wir mit jeder dieser $n-1$ oder n Normformen so lange, bis wir entweder mindestens die entsprechende Anzahl an Relationen gefunden haben oder bis wir selbst bei einer Verdopplung der Größe des Siebintervalls keine neue Relation mehr finden. Damit haben wir auch schon das nächste Teilproblem angesprochen, nämlich die Bestimmung des Siebintervalls. Das Siebintervall muss in unserer Situation angesichts verschiedener automatisch generierter Polynome offensichtlich ebenfalls automatisch bestimmt werden, während man im originalen Number–Field–Sieve damit zufrieden sein kann, nachdem man nach einigem Probieren ein passendes Polynom gefunden hat, das Siebintervall manuell und mit natürlicher Intelligenz zu bestimmen. Diesen Punkt wollen wir allerdings zunächst noch zurückstellen und zunächst weitere Variationsmöglichkeiten für das Polynom betrachten.

5.7 Sieben in einem Ideal

Nun wenden wir uns einem etwas komplizierteren Problem zu. In der Praxis zeigt sich, dass es häufig zu viele lineare Abhängigkeiten zwischen den Relationen gibt, bzw. dass manche Primideale in keiner Relation vorkommen. Die Lösung, die wir dafür entwickelt haben, basiert auf der Idee des vorangegangen Abschnitts, nur stellen wir nun eine zusätzliche Randbedingung an die Zahlen α und β, die wir wählen.

Um zu erzwingen, dass ein festes Primideal $\mathfrak{p}$ in einer Relation vorkommt, müssen wir eine algebraische Zahl γ in diesem Primideal finden, die die Eigenschaft hat, dass das von ihr erzeugte Hauptideal über der Faktorbasis zerfällt. Dann ist nämlich wegen $(\gamma) \subseteq \mathfrak{p}\,\mathfrak{p}$ ein Teiler von (γ) (vgl. Definition 1.3.6) und kommt somit in der Primidealzerlegung vor. Eine notwendige Bedingung für γ ist dabei, dass die Norm zerfällt. Wenn wir nun α und β so wählen, dass α,$\beta \in \mathfrak{p}$ gilt, dann gilt auch $a\alpha + b\beta \in \mathfrak{p}$ für alle $a, b \in \mathbb{Z}$ und wir können das Polynom $\mathcal{N}(x\alpha + y\beta)$ betrachten. Da nun $\mathfrak{p}$ jedes $(x\alpha + y\beta)$ teilt, zeigt sich dabei, dass jeder Koeffizient dieser Normform durch $\mathcal{N}(\mathfrak{p})$ teilbar ist. Daher dividieren wir

jeden Koeffizienten durch $\mathcal{N}(\mathfrak{p})$. Nun können wir wie zuvor versuchen, mit dem Siebalgorithmus Stellen (x, y) zu finden, an denen dieses Polynom über unserer Faktorbasis zerfällt. Wenn man anschließend die Idealfaktorisierung bestimmt, muss man beachten, dass man noch den zusätzlichen Faktor $\mathfrak{p}$ benötigt, durch dessen Norm die Koeffizienten dividiert wurden.

Solche α und β können wir finden, indem wir die Konjugiertenvektoren einer Basis des Ideals bestimmen und dann in bereits bekannter Weise möglichst kurze Vektoren in diesem Gitter bestimmen.

Zu jedem Primideal finden wir auf diese Weise ein Paar (α, β) und ein Polynom. In der Praxis zeigt sich, dass dieses Polynom häufig immer noch zu große Koeffizienten hat, daher suchen wir nach Variationsmöglichkeiten, um jeweils mehrere Polynome anbieten zu können. Dabei zeigt sich zunächst, dass in dem soeben beschriebenen Verfahren nirgendwo benötigt wird, dass $\mathfrak{p}$ ein Primideal ist. Daher können wir statt $\mathfrak{p}$ auch ein $\mathfrak{a}$ verwenden, das wir aus $\mathfrak{p}$ erhalten, indem wir ein zufälliges Potenzprodukt aus Primidealen kleiner Norm an $\mathfrak{p}$ multiplizieren, d.h. wir betrachten nun

$$\mathfrak{a} = \mathfrak{p} \prod_{i=1}^{l} \mathfrak{p}_i^{e_i},$$

wobei l eine feste kleine Zahl (etwa in der Größenordnung von 10) und jedes e_i eine kleine Zufallszahl (etwa aus dem Bereich von -4 bis 4) ist. Damit haben wir für jedes Primideal nun sehr viele verschiedene Polynome zur Auswahl (mit den angegebenen Zahlen 9^{10}). Diese Funktionalität wird von der Funktion `rel_gen` zur Verfügung gestellt, der man im Parameter `missing_ideal` übergibt, für das wievielte Primideal der Faktorbasis man sich gerade interessiert, die weiteren Parameter haben dieselbe Bedeutung wie für die Funktion `sieve`, welche automatisch aufgerufen, wobei dieser im Parameter `ideal_to_divide_by` das Ideal $\mathfrak{p}$ mit übergeben wird.

5.8 Kombination mit der Reduktionstheorie

Für Primideale $\mathfrak{p}$ mit großer Norm erhalten wir allerdings mit dem soeben beschriebenen Verfahren immer noch Polynome mit zu großen Koeffizienten. Insbesondere sind die Koeffizienten häufig wesentlich größer, als die Koeffizienten des Körperpolynoms. Daher haben wir noch eine weitere Modifikation entwickelt und implementiert, die diesem Problem abhelfen soll und die sich in der

Praxis glänzend bewährt. Dazu kombinieren wir die Siebverfahren mit der in Kapitel 5.2 beschriebenen Reduktionstheorie.

An Stelle des Ideals

$$\mathfrak{a} = \mathfrak{p} \prod_{i=1}^{l} \mathfrak{p}_i^{e_i}$$

berechnen wir unmittelbar ein dazu äquivalentes reduziertes Ideal, indem wir wie beschrieben nach jeder Operation eine Reduktion vornehmen, d.h. wir erhalten nun $\mathfrak{a}' = \mathfrak{a}/(\delta)$. Nun suchen wir statt Zahlen $\alpha, \beta \in \mathfrak{a}$ mit kleiner Norm Zahlen $\alpha', \beta' \in \mathfrak{a}'$ mit kleiner Norm. Da $\mathfrak{a}'$ eine Norm hat, die wesentlich kleiner ist als die Norm von $\mathfrak{a}$, erhalten wir nun aus $\mathcal{N}(x\alpha' + y\beta')$ auch ein Polynom mit kleineren Koeffizienten, jedoch ist nun scheinbar $\mathfrak{a}$ und damit $\mathfrak{p}$ nicht mehr an den gefundenen Relationen beteiligt. Diesem Mangel ist jedoch leicht abzuhelfen. Haben wir nämlich eine Relation

$$(x\alpha' + y\beta') = \mathfrak{a}' \cdot \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}$$

gefunden, so ist

$$((x\alpha' + y\beta') \cdot \delta) = \mathfrak{a} \cdot \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}.$$

Damit lässt sich unser Verfahren zur Erzeugung von Relationen in denen ein festes Ideal $\mathfrak{a}$ vorkommt, wie folgt formalisieren:

5.8.1. Algorithmus

Sieben in einem Ideal

EINGABE: Eine Faktorbasis $FB = \{\mathfrak{p}_1, \ldots, \mathfrak{p}_k\}$ und ein Ideal $\mathfrak{a} = \prod_{i=1}^{k} \mathfrak{p}_i^{f_i}$
AUSGABE: Relationen der Form $(\alpha) = \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}$ mit $e_i \geq f_i$

(1) Reduziere $\mathfrak{a}$ zu $\mathfrak{a}' = \mathfrak{a}/(\delta)$
(2) **repeat**
(3) Bestimme $\alpha, \beta \in \mathfrak{a}'$ mit kurzen Konjugiertenvektoren
(4) Bestimme das Polynom $f(x, y) := \mathcal{N}(x\alpha + y\beta)$ (etwa durch Berechnung von $n + 1$ Normen und Interpolation)
(5) Dividiere $f(x, y)$ durch $\mathcal{N}(\mathfrak{a}')$
(6) Finde mittels NFS glatte Stellen von $f(x, y)$
(7) **for** (jedes Kandidatenpaar (x, y)) **do**
(8) **if** ($(x\alpha + y\beta)/\mathfrak{a}'$ zerfällt über der Faktorbasis) **then**
(9) Schreibe die Zerlegung $(x\alpha + y\beta)/\mathfrak{a}' = \prod_{i=1}^{k} \mathfrak{p}_i^{g_i}$
(10) Speichere die Relation $(x\alpha + y\beta) \cdot \delta = \prod_{i=1}^{k} \mathfrak{p}_i^{g_i + f_i}$
(11) **fi**
(12) **od**
(13) **if** (Nicht ausreichend Relationen gefunden) **then**
(14) Multipliziere $\mathfrak{a}$ mit einen zufälligen Potenzprodukt von Faktorbasiselementen kleiner Norm, führe dabei nach jeder Multiplikation eine Reduktion durch und passe den Vektor $(f_1, \ldots, f_k)$ entsprechend an.
(15) **fi**
(16) **until** (genügend Relationen gefunden)

Dieser Algorithus wird in der Funktion `rel_gen_reduction` implementiert, wobei die Parameter dieselben sind, wie für die Funktion `rel_gen`.

5.9 Wahl des Siebintervalls

Im Gegensatz zum Number–Field–Sieve, wo typischerweise pro zu faktorisierender Zahl ein Polynom bestimmt wird, das zum Sieben benutzt wird, benutzen

wir zur Bestimmung der Klassenzahl mehrere, womöglich sogar sehr viele Polynome. Daher ist es nicht mehr praktikabel, für jedes Polynom vom Benutzer ein Siebintervall festlegen zu lassen, wie dies beim Number–Field–Sieve typischerweise passiert. Durch die möglicherweise stark unterschiedlich großen Koeffizienten ist es aber auch ausgeschlossen, einfach für alle Polynome dasselbe Siebintervall zu verwenden. Es bleibt also nicht anderes übrig, als eine Heuristik zu implementieren, die für ein fixiertes Polynom jeweils ein geeignetes Siebintervall auswählt.

Diese Heuristik lehnten wir mangels spezifischer Erfahrungen an die typische Vorgehensweise zur Konstruktion geeigneter Siebintervalle für das Number–Field–Sieve an. Daher wählen wir Siebintervalle der Form $[-a, a] \times [1, b]$, d.h. wir sieben jeweils in Rechtecken die symmetrisch zur y–Achse liegen.

Um dabei eine Abschätzung für die benötigte Intervallgröße zu erhalten, benutzen wir zu Beginn des Algorithmus ein sehr kleines Intervall, auf dem wir die Relationen für die Polynome aus Abschnitt 5.6 bestimmen, um dann zu sehen, wieviele Paare (x, y) wir im Siebintervall durchschnittlich betrachten müssen, um eine Relation zu finden. Um hier wirklich korrekte Zahlen zu erhalten, müsste man natürlich ein repräsentatives Siebintervall betrachten. Da es sich jedoch um eine Vorberechnung handelt, wollen wir hier keinen allzu hohen Aufwand treiben, so dass wir uns mit sehr groben Annäherungen zufrieden geben. Speziell für die Verfahren zum Sieben in einem Ideal können wir dabei diesen Wert sukzessive der Realität annähern, da wir hier meist Polynome mit ähnlichem Verhalten finden.

Nun müssen wir noch die Grenzen a und b etwas verfeinern. Dazu wollen wir a und b so optimieren, dass das Polynom auf diesem Intervall möglichst kleine Werte annimmt. Dabei erweist es sich in der Praxis als gangbarer Weg, das Polynom f jeweils an möglichen Eckpunkten des Intervalls zu evaluieren und diese Werte zu vergleichen. Dabei müssen wir auch berücksichtigen, dass mit wachsendem b häufiger die Initialisierungsschritte durchgeführt werden müssen, die für jede neue Zeile des Siebintervalls anfallen. Daher ist es günstiger, auch etwas größere Funktionswerte in Kauf zu nehmen, wenn dafür der Wert von b niedriger gehalten werden kann. In unserer Implementierung geht daher ein zusätzlicher Gewichtsfaktor ein, der dieser Überlegung Rechnung tragen soll.

In der Praxis testen wir daher, ob $4\,|f(a/2, 2b)| < |f(a, b)|$ gilt, und ersetzen (a, b) nur dann durch $(a/2, 2b)$, wenn diese Bedingung erfüllt ist, während wir umgekehrt (a, b) durch $(2a, b/2)$ ersetzen, sobald $|f(a * 2, b/2)| < 4\,|f(a, b)|$

gilt. All dies passiert implizit innerhalb der Funktion `sieve`. Dabei erhalten wir typischerweise Siebintervalle, die im Vergleich zu den im Number–Field–Sieve verwendeten Intervallen extrem „flach“ und „lang“ sind, d.h. wir erhalten sehr große Werte für a und sehr kleine Werte für b, im praktischen Vergleich mit von Hand gewählten Intervallen, wie sie für das Number–Field–Sieve eher Verwendung finden würden, erweist sich diese Gestalt der Siebintervalle aber insofern als korrekt, als wir damit deutlich schneller viele Relationen finden.

Dabei sind die Werte für a in der Tat so groß, dass wir an die Grenzen der ursprünglichen Implementierung des Siebtools stießen, das implizit voraussetzt, dass $a <$ `MAXINT`$/4$ gilt, wobei mit `MAXINT` die maximale Zahl bezeichnet wird, die der Rechner noch als vorzeichenbehaftete Integerzahl darstellen kann. Bei der verwendeten (32-Bit-) Hardware musste also $a < 2^{29}$ gewählt werden. Durch Verwendung des Datentyps `unsigned int` und einige Optimierungen an Stellen, an denen arithmetische Operationen auf den Intervallgrenzen ausgeführt wurden, gelang es uns relativ leicht, das Siebtool dahingehend zu optimieren, dass es nun bei gleicher Effizienz möglich ist, $a <$ `MAXINT` zu wählen. Selbst dies erweist sich bei Beispielberechnungen in Zahlkörpern mit großer Diskriminante noch als unangenehme Einschränkung. Da jedoch eine Überwindung dieser Schranke einerseits mit hohem Programmieraufwand verbunden ist und andererseits in diesem Falle auch ein deutlicher Verlust an Effizienz zu befürchten wäre (etwa durch Verwendung einer Langzahlarithmetik), haben wir diese Beschränkung im Programm belassen.

5.10 Vereinfachung der Relationenmatrix

Nachdem wir die Methoden zum Auffinden von Relationen beschrieben haben, wollen wir uns nun der Frage zuwenden, wie wir die gefundenen Relationen weiter verarbeiten. Wie wir schon andeuteten, liegt eine besondere Schwierigkeit des Verfahrens darin, dass der Schritt der linearen Algebra, der nötig ist, um die Struktur der Klassengruppe zu bestimmen, in $\mathbb{Z}$ stattfindet und nicht in einem endlichen Körper, wie man es etwa von Faktorisierungsalgorithmen und Algorithmen zur Berechnung des diskreten Logarithmus in endlichen Körpern her gewohnt ist. Genauer müssen wir, um später in der Klassengruppe rechnen zu können, die Hermite–Normalform der Relationenmatrix A bestimmen, d.h. eine obere Dreicksmatrix H, die durch eine unimodulare Transformation aus A hervorgeht und in der alle Einträge rechts eines Diagonalelements modulo

des Diagonalelements reduziert sind.

Da dieser Algorithmus vergleichsweise (d.h. im Vergleich zum Lösen von Gleichungssystemen modulo 2 bzw. p) langwierig ist, ist es hier von besonderer Bedeutung, schon vor dem Start der Berechnung möglichst eine Vereinfachung der Relationenmatrix vorzunehmen.

Dabei können wir insbesondere Primideale, die nur in einer Relation vorkommen und dort den Exponenten 1 haben, aus der Faktorbasis streichen und somit die korrespondierende Zeile und Spalte aus der Matrix löschen, da ein solches Ideal offenbar keinen Beitrag zur Erzeugung der Klassengruppe leistet. Darüber hinaus können wir Relationen löschen, in denen nur ein Primideal mit Exponent 1 vorkommt, da eine solche Relation nur sagt, dass dieses Ideal ein Hauptideal ist. Auch in diesem Fall können wir dann das Primideal und die entsprechende Zeile in der Relationenmatrix löschen. Allerdings müssen wir in diesem Fall darauf achten, die Erzeuger in den verbleibenden erweiterten Relationen entsprechend anzupassen.

Im Laufe dieser Überprüfungen suchten wir zunächst auch nach linear abhängigen Zeilen in der Relationenmatrix, um dann für die entsprechenden Ideale mit Algorithmus 5.8.1 zusätzliche Relationen zu erzeugen. So konnten wir garantieren, dass die Matrizen, deren Hermite–Normalform wir anschließend berechnen wollten, auch vollen Rang haben. Dieses Verfahren fand in den Beispielen aus Kapitel 5.13 Verwendung. Inzwischen wird diese Aufgabe jedoch von den entsprechend adaptierten Routinen zur Hermite–Normalform–Berechnung übernommen.

Dieser Teil des Verfahrens ist in der Funktion `simplify` implementiert.

5.11 Berechnung der Hermite–Normalform

Die Berechnung der Hermite–Normalform großer, dünn besetzter Matrizen wird in der Arbeit von Patrick Theobald ([The00]) näher untersucht und beschrieben, so dass wir uns hier auf eine kurze Beschreibung der grundlegenden Strategie beschränken. Zunächst startet die Normalform–Berechnung mit Operationen in $\mathbb{Z}$. Dabei wird das Pivotelement, d.h. dasjenige Element, mit dessen Hilfe die restlichen Elemente einer Zeile eliminiert werden, so gewählt, dass die Matrix einerseits möglichst dünn besetzt bleibt und andererseits die durch Linearkombinationen entstehenden neuen Einträge der Matrix (betragsmäßig) möglichst klein bleiben. Mit dieser Strategie arbeitet man so lange, bis die verbleibende

Matrix entweder dicht besetzt ist oder die Einträge gar zu groß sind. Dann setzt man die Berechnung mit modularen Methoden fort, etwa basierend auf den Ideen aus [DKTJ87]. Dabei zeigt sich in der Praxis, dass häufig die Berechnung (eines Vielfachen) der Gitterdeterminante, die von diesen Verfahren benötigt wird, das Hauptproblem darstellt, obwohl hierbei der chinesische Restsatz angewendet werden kann. Die oberen Schranken, die man für das Vielfache der Gitterdeterminanten angeben kann, das man gerade berechnet, sind nämlich im Normalfall extrem groß. Hier lassen sich allerdings erhebliche Verbesserungen erzielen, wie in [The00] gezeigt wird.

Ein zur Zeit noch offenes Problem ist, wie man am besten verfährt, um bei all diesen Transformationen der Ideale die Erzeuger anzupassen.

5.12 Verifikation des Resultats

Beim Faktorisieren und bei der Berechnung des diskreten Logarithmus sind wir im wesentlichen fertig, sobald das Problem der Linearen Algebra gelöst ist. Beim Faktorisieren etwa haben wir schlimmstenfalls eine triviale Lösung der Art $N = 1 \cdot N$ gefunden. Indem wir stets mehrere Lösungen zugleich berechnen, können wir in der Praxis davon ausgehen, dass wir stets wenigstens eine nicht triviale Lösung erhalten.

Im Gegensatz dazu bleiben nach Berechnung der Hermite–Normalform zur Klassengruppenberechnung noch zwei offene Probleme übrig. Zunächst hat die Matrix, deren HNF wir berechnet haben, im allgemeinen nicht vollen Zeilenrang. Im Falle der Faktorisierung bedeutet das, dass ein bestimmtes Faktorbasiselement keinen Beitrag dazu geleistet hat, eine Faktorisierung zu finden - dies ist also völlig unkritisch. In unserem Szenario bedeutet dies jedoch, dass wir nicht wissen, ob dieses Primideal einen Beitrag zur Klassengruppe liefert, oder nicht. Das andere offene Problem ist, dass wir, selbst wenn die Dimension des Gitters stimmt, immer noch nicht wissen, ob unsere Relationen wirklich das volle Relationengitter erzeugen, oder ob wir nur ein Teilgitter gefunden haben, d.h. wir wissen a priori nicht, ob die Relationenmatrix wirklich die Klassenzahl als Determinante hat, oder ob wir nur ein Vielfaches der Klassenzahl gefunden haben.

Das erste dieser beiden Probleme ist leicht zu lösen: Mit Algorithmus 5.8.1 oder mit der Reduktionstheorie aus Kapitel 5.2 bestimmen wir dann für jedes Primideal in der Faktorbasis, für das wir noch nicht wissen, ob es einen Beitrag

zur Klassengruppe liefert, (mindestens) eine Relation. Dies geschieht mit der Funktion `cgr_finish`, der die aktuelle Relationenmatrix und die Indizes linear abhaengiger Zeilen uebergeben werden. Nachdem dieses Verfahren abgeschlossen ist, wird erneut die Hermite–Normalform berechnet. Dies ist nun sehr leicht möglich, da die Matrix schon „fast“ Hermite–Normalform hat. Sollten wir noch immer keine Matrix mit vollem Rang haben, wiederholen wir diesen Schritt solange, bis voller Rang erreicht ist.

Mit einer ähnlichen Methode ist allerdings auch das zweite Problem zu behandeln, dass man auf jeden Fall hat: Wir müssen verifizieren, ob wir nur ein Vielfaches der Klassenzahl gefunden habent. Dazu kann man die analytische Klassenzahlformel (Satz 1.3.13) verwenden. Hierzu bestimmt man zunächst die Zahl der Einheitswurzeln im Zahlkörper (etwa mit Hilfe von Algorithmus 4.9.9 aus [Coh95]), sowie eine ausreichend präzise Approximation an

$$\prod_{p \in \mathbb{P}} \frac{1 - \frac{1}{p}}{\prod_{\mathfrak{p} \text{ über } p \text{ prim}} 1 - \frac{1}{\mathcal{N}(\mathfrak{p})}}.$$

Nun muss man noch den vermuteten Regulator bestimmen, wozu man die Determinante des Gitters ausrechnen muss, das von den Logarithmenvektoren der Idealerzeuger erzeugt wird, deren Exponentenvektoren im Laufe der Hermite–Normalform–Berechnung zu 0 reduziert wurden. Nähere Untersuchungen über dieses Teilproblem werden für Körpergrad 2 in [Mau00] angestellt. Im allgemeinen Fall steht eine Implementierung in LiDIA noch aus. Daher greifen wir für unsere Zwecke auf eine bereits existierende Implementierung zurück: Wir benutzen zur Berechnung des Regulators die Funktion, die PARI in seiner Oberfläche `gp` zur Verfügung stellt, und übergeben unserem Algorithmus interaktiv diesen Wert. Dabei stoßen wir bei größeren Beispielen allerdings auch dort an die Grenzen der Implementierung, so dass wir in diesen Fällen keinen Regulator angeben können. In diesem Fall wissen wir dann nur, dass wir ein Vielfaches der Klassenzahl gefunden haben. Für einige Anwendungen in der Praxis ist auch dies ausreichend.

Andernfalls können wir den Wert von $h \cdot R$ mit dem Näherungswert vergleichen, den man aus der Approximation A an die analytische Klassenzahlformel erhält. Stimmen diese Werte hinreichend gut überein, d.h. gilt $\frac{hR}{A} < \sqrt{2}$, so hat man Klassengruppe und Regulator gefunden, andernfalls müssen weitere Relationen erzeugt werden. Dabei erweist sich der naive Ansatz, der Reihe nach für die Ideale, die die Klassengruppe erzeugen, jeweils mit Algorithmus 5.8.1 einige Relationen zu erzeugen, in der Praxis als wenig erfolgreich. Dies ist aber auch

nicht überraschend, wenn man sich klarmacht, dass nun für ein Ideal $\mathfrak{q}$, das z.B. einen 10-stelligen Faktor zur Klassenzahl beiträgt, Relationen der Form

$$(a + b\rho) = \mathfrak{q} \cdot \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}$$

gesucht werden.

Eine cleverere Strategie besteht darin, von der vorhandenen Relation der Form

$$(a + b\rho) = \mathfrak{q}^e \cdot \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}$$

ausgehend für kleine Teiler x von e auszuprobieren, ob man Relationen der Form

$$(a + b\rho) = \mathfrak{q}^{e/x} \cdot \prod_{i=1}^{k} \mathfrak{p}_i^{e_i}$$

finden kann, d.h. statt Relationen in $\mathfrak{q}$ zu suchen, sucht man Relationen in $\mathfrak{q}^{e/x}$. Dabei erhält man aus der Approximation an das Produkt aus Klassenzahl und Regulator auch eine obere Schranke für x. Dies wird in der Funktion `cgr_improve` implementiert, der neben der Relationenmatrix diese obere Schranke für x übergeben wird.

Dabei ist es von entscheidender Bedeutung, nicht für einen festen Wert von x und ein festes Ideal $\mathfrak{q}$ zu viel Mühe zu investieren, sondern die Anstrengungen auf alle Kandidaten gleichmäßig zu verteilen. Für jede gefundene Relation überprüfen wir schließlich, ob sich Regulator oder Klassenzahl ändern, und korrigieren die Schranke für x gegebenenfalls entsprechend nach unten.

Ein Vergleich mit den Resultaten anderer Strategien zur Relationengenerierung zeigt, dass beide Probleme in unserem Szenario besonders häufig auftreten. Es ist zu vermuten, dass dies durch die systematische Relationengenerierung durch Siebverfahren bedingt ist, da es gerade die stärker zufallsbasierten Algorithmen sind, die hier weniger Probleme haben. Dies trifft sowohl für die in [BD91] implementierte Methoden der Relationengenerierung (im wesentlichen Probedivision reduzierter zufälliger Idealprodukte über der Faktorbasis) als auch für die in [JJ98] beschriebene Implementierung zu, die zwar auch Siebverfahren verwendet, jedoch sehr viel mehr verschiedene (zufällige) Polynome verwendet als wir dies tun, da im Spezialfall des quadratischen Siebs die Initialisierung pro Polynom wesentlich einfacher ist.

5.13 Erste Ergebnisse

Mit den bisherigen Ideen lässt sich schon eine vergleichsweise effiziente Implementierung erreichen. In der Praxis haben wir hier eine parametrisierte Familie von Zahlkörpern untersucht, die von H.-J. Stender bereits genauer studiert wurde ([Ste77]). In seinen Arbeiten hat Stender insbesondere gezeigt, wie man formelmäßig die Fundamentaleinheiten dieser Körper angeben kann. Damit können wir den Regulator des Zahlkörpers berechnen und haben eine wesentliche Erleichterung bei der Lösung des zweiten Problems, das wir im vorherigen Abschnitt beschrieben haben.

Die untersuchten Zahlkörper zeichnen sich dadurch aus, dass das Körperpolynom von der Form $f(x) = x^3 - (D^3 + d)$ ist, wobei D natürliche Zahlen bezeichnet und $d \in \{-1, +1\}$ gilt. In [Ste77] wird zwar sehr viel allgemeiner nur $d|D^2$ und d kubenfrei vorausgesetzt, da wir aber auch am Verhalten für wachsende Diskriminanten interessiert sind, ist es für uns uninteressant, viele verschiedene Werte von d zu benutzen.

Für einige Körper dieser Bauart haben wir zunächst die Struktur der Klassengruppe (Tabelle 5.1) und dann die Laufzeiten verschiedener Berechnungsteile und anderer Implementierungen (Tabelle 5.2) zusammengestellt.

Zu den Versuchen, mit KaSh die Klassengruppen dieser Körper zu berechnen, ist zu sagen, dass bei der vorgegebenen Faktorbasisgröße der vorhandene Hauptspeicher von 256 MB nicht mehr ausreichte, ein Ergebnis zu ermitteln – hier zeigt sich deutlich der Vorteil unserer speziellen Darstellung von dünn besetzten Matrizen. Darüber hinaus kann man anhand der beiden Beispiele, die wir mit KaSh erfolgreich berechnet haben, vermuten, dass unser Siebverfahren wie erwartet deutliche Effizienzvorteile hat, sobald viele Relationen gefunden werden müssen.

5.14 Weitere Beispiele

Nun wollen wir uns allgemeineren Beispielen zuwenden. Dazu wählen wir Serien von zufälligen Polynomen mit einer festen Zahl von reellen Einbettungen und wachsender Diskriminante. Dabei kennen wir insbesondere die Regulatoren dieser Körper nicht, so dass wir nun die oben skizzierten zusätzlichen Aufgaben zu behandeln haben.

Dabei wollen wir zunächst zeigen, wie sich der Zeitbedarf für die Siebpha-

D	d	Δ	R	h	|FB|	lin.
11	-1	-47760300	5.89...	[3,3,3,6]	247	2
1042	-1	-3839967668 441074707	14.99...	[3,3,2114778]	2413	99
1042	$+1$	-3455970913 8157209867	14.99...	[3,3,9746730]	2663	136
4988	-1	-4158363335 6304901015 4907	18.12...	[3,3,3,3,3, 38452158]	3749	170
4988	$+1$	-4620403706 4050227027 587	18.12...	[3,3, 215710230]	3552	134
47520	-1	-3454457441 1926853892 349952003	22.63...	[3,3,3,6, 13597799064]	5382	91
226082	-1	-3605451707 0224707505 5401305965 0603	25.75...	[3,3,3,3,3,3,27, 20846798199]	6974	127
226082	$+1$	-4006057452 2471911095 2722293930 483	25.75...	[3,3,3,3,3,6, 111510282834]	6742	108
1075464	-1	-4641907898 9374092480 2377944493 2178947	28.87...	[6,6, 693173937012264]	8461	161
1075464	$+1$	-4641907898 9374092629 5065383808 6147075	28.87...	[3,3,3,3, 283185632461968]	8426	161

Tabelle 5.1: Klassenzahlen für einige Zahlkörper

D, d: definieren den Zahlkörper wie im Text beschrieben, Δ: Körperdiskriminante, R: Regulator, h: Klassenzahl der zugehörigen Zahlkörpers als Produkt der Größen der zyklischen Untergruppen, |FB|: Größe der Faktorbasis, lin.: Zahl der linear abhängigen Zeilen in der Relationenmatrix.

D	d	T_s	T_H	T_r	T_L	T_K
11	-1	21s	1s	0s	22s	5s
1042	-1	8m	19m	4.5m	31.5m	1h46m
1042	$+1$	9m	35m	7m	51m	–
4988	-1	18m	1h21m	12m	1h51m	–
4988	$+1$	15m	62m	9m	1h26m	–
47520	-1	61m	2h23m	10m	3h34m	–
226082	-1	42m	6h21m	20m	7h23m	–
226082	$+1$	44m	5h16m	16m	6h16m	–
1075464	-1	68m	11h21m	30m	13h	–
1075464	$+1$	60m	12h15m	30m	13h45m	–

Tabelle 5.2: Laufzeiten zur Berechnung einiger Klassengruppen

D,d: definieren den Zahlkörper wie im Text beschrieben T_s: Zeit für die Siebphase in unserer Implementierung, T_H: Zeit für die HNF-Berechnung in unserer Implementierung, T_r: Zeit für das Erreichen des vollen Rangs in unserer Implementierung, T_L: Gesamtdauer der Berechnung mit unserer Implementierung (LiDIA), T_K: Dauer der Berechnung mit KaSh.

se entwickelt. Global können wir zunächst feststellen, dass der Zeitaufwand mit steigendem Polynom- bzw. Körpergrad und mit steigender Diskriminante natürlich ansteigt. Dabei ist die Auswahl des Siebintervalls auf den Bereich von ganzen Zahlen, die sich mit dem Datentyp `int` darstellen lassen, eingeschränkt. Dadurch passiert es mit steigendem Körpergrad immer schneller, dass wir nicht mehr genügend Relationen finden können, da die Polynomwerte in dem möglichen Siebintervall zu groß werden.

Um diesem Effekt Rechnung zu tragen, vergrößern wir die Faktorbasis in Abhängigkeit vom Körpergrad noch über die durch die Bachschranke vorgegebene Mindestgröße hinaus.

Die Siebzeiten haben wir in Anhang A tabelliert.

Als nächste Aufgabe haben wir dann die HNF der Relationenmatrizen zu berechnen. Um den Regulator bestimmen zu können, benötigen wir einige Elemente aus dem Kern der ursprünglichen Relationen-Matrix, denn Elemente des Kerns entsprechen bei Berechnung des Potenzprodukts der Erzeuger gerade Einheiten. Da in LiDIA eine Implementierung für dieses Teilproblem aber nicht existiert, verwenden wir an dieser Stelle gp, um den Regulator auszurechnen.

Für unsere Beispiele haben wir in Anhang B die Zeiten für die Berechnung

der HNF, sowie die Ergebnisse für Regulator und Klassenzahl tabelliert. In den Fällen, in denen kein Regulator angegeben ist, konnten wir diesen mit den Mitteln von gp nicht bestimmen. Insbesondere kennen wir in diesen Fällen also auch nur ein Vielfaches der Klassenzahl, von dem wir nicht wissen, ob es bereits die Klassenzahl selbst ist. Schließlich bestimmen wir noch einige zusätzliche Relationen, um vollen Zeilenrang der Relationenmatrix zu gewährleisten und um im Falle, dass wir den Regulator bestimmen konnten, gegebenenfalls die echte Klassenzahl (und nicht nur ein Vielfaches davon) zu finden. Die Laufzeiten hierfür sind ebenfalls in Anhanf B aufgeführt.

Ein Beispiel mit den detaillierten Informationen, die im Laufe der Berechnung ausgegeben werden, haben wir in Anhang C wiedergegeben und kommentiert.

5.15 Bewertung und Ausblick

Wenn wir einen Vergleich mit den bisherigen Einsatzgebieten des Number–Field–Sieves, der Faktorisierung und der Berechnung diskreter Logarithmen in $\mathbb{Z}/p\mathbb{Z}$, anstellen, so zeigt sich, dass sich einige Veränderungen an der Siebstrategie ergeben. Diese sind vor allem durch die Einschränkungen bedingt, denen wir bei der Auswahl der Polynome unterliegen. So spielt zum Beispiel bei den anderen Anwendungsgebieten inzwischen das Zerfallsverhalten der kleinen Primzahlen eine wichtige Rolle für die Auswahl des Polynoms, das für die weiteren Berechnungen benutzt wird. In unserer Situation ist dieses Verhalten jedoch durch die Auswahl des Zahlkörpers im wesentlichen vorgegeben.

Darüber hinaus benötigen wir im Gegensatz zu den anderen Anwendungsgebieten Relationenmatrizen mit vollem Zeilenrang. Um dies zu gewährleisten verwenden wir das Verfahren aus Kapitel 5.7. Als wesentlicher Unterschied zu den bisherigen Einsatzgebeiten des Number–Field–Sieves ergibt sich also, dass wir nun Polynome und die jeweiligen Siebintervalle dynamisch konstruieren.

Die Effizienz der hierzu entwickelten Heuristiken können wir dadurch belegen, dass es mit unserem Verfahren tatsächlich möglich ist, sehr schnell viele Relationen zu finden. In der Praxis haben wir das Verfahren für „zufällige“ Zahlkörper der Grade 3 bis 6 bis hin zu etwa 50-stelligen Diskriminanten anwenden können.

Will man darüber hinaus noch größere Determinanten bearbeiten, so erweisen sich weniger die Heuristiken als vielmehr die Randbedingungen, unter

denen das Siebtool ursprünglich entwickelt wurde, als Problem. Abgesehen von der Einschränkung, dass das Siebintervall, das wir automatisch auswählen, stets von der Form $[-a, a] \times [1, b]$ ist, müssen die Werte a und b vor allem stets im Bereich des Datentyps `int` liegen. Bei grösseren Determinanten liegen aber die Nullstellen der verwendeten Polynome typischerweise außerhalb dieses Bereichs, so dass auch ein aussichtsreiches Sieben außerhalb des `int`–Bereichs stattfinden müsste.

Anhang A

Laufzeiten für die Siebphase unseres Verfahrens

In den folgenden Tabellen ist jeweils f das Körperpolynom, i der Index der Gleichungsordnung in der Maximalordnung, Δ die Diskriminante, $p_{\max}$ die maximale (rationale) Primzahl, deren Primideale Elemente der Faktorbasis sind, T_f die Zeit zur Faktorisierung aller Primzahlen in Primideale, $a \times b$ bezeichnet das initiale Siebintervall, das wir stets als $[-a, a] \times [1, b]$ wählen, T_1 bezeichnet die Zeit, die wir mit dem Sieben über dem Siebintervall für die initial bestimmten Polynome verbringen, T_2 ist die Zeit, die wir darüber hinaus aufwenden, um weitere Relationen zu finden, T_3 ist die Zeit für den initialen Vereinfachungsschritt der Matrix.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^3 + -46 * x^2 + -5 * x + -4$	1	-1520968	261	0s	2128×1	2s	0s	0s
$x^3 + 180 * x^2 + -664 * x + 1587$	1	-25047695507	2579	1s	92423×1	18s	0s	0s
$x^3 + -52240 * x^2 + 57907 * x + -39659$	1	-13463382674179299599	8730	4s	4234413×1	1m 1s	0s	5s
$x^3 + -2112423 * x^2 + 3622665 * x + -4396457$	1	-10720711124487377981 4835404	16165	8s	58074558×1	2m 55s	0s	17s
$x^3 + 158148413 * x^2 + 34308993 * x + 186251382$	3	-32415357485157562818 6556629939403	25217	16s	1073741823×1	6m 6s	0s	28s
$x^3 + 4133951584 * x^2 + 6810847149 * x + 8211876627$	1	-15278466535970895911 88139605048420071079	36632	19s	2147483647×1	17m 27s	0s	1m 3s
$x^3 + 333512969279 * x^2 + 426535438826 * x + 394946209572$	2	-95921705643103658494 06319702394376391086 451851	50444	32s	failed			

Tabelle A.1: Körper vom Grade 3 mit einer reellen Einbettung.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^3 + -44 * x^2 + 21 * x + 35$	1	12127297	1197	0s	21043×1	5s	0s	0s
$x^3 + 246 * x^2 + -1377 * x + -1210$	6	5682807261	2270	1s	71560×1	13s	0s	0s
$x^3 + -10250 * x^2 + 50113 * x + 41948$	4	27727946219554744	6450	4s	2311518×1	46s	0s	2s
$x^3 + -3318573 * x^2 + -3824415 * x + 1448222$	1	37279111861718358458 2948773	16845	9s	63056450×1	3m 4s	0s	14s
$x^3 + 74064508 * x^2 + -145225378 * x + -131571154$	1	32951393401334176688 9424492650772	25228	13s	536870911×1	5m 57s	0s	39s
$x^3 + 3822941529 * x^2 + 4286379226 * x + 240688561$	1	21472906031672211421 7718092862652948121	35056	18s	2147483647×1	13m 51s	0s	1m 4s
$x^3 + 88361718083 * x^2 + -352726420334 * x + 311810143202$	1	11093006330881910439 37616182577255056733 77304	46284	24s	2147483647×1	33m 34s	0s	1m 39s
$x^3 + 8110566983176 * x^2 + -16141692821563 * x + -5841054185390$	4	18503098507274685820 64772520318826609977 505413000972	62708	39s	failed			

Tabelle A.2: Körper vom Grade 3 mit 3 reellen Einbettungen.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^4+-3*x^3+6*x^2+-10*x+18$	1	497556	26	0s	169×1	0s	0s	0s
$x^4+245*x^2+266*x+191$	3	767965702576	5991	5s	39436×1	1m 4s	0s	3s
$x^4+-83*x^3+5518*x^2+2991*x+4956$	3	5385529680968789313	14881	13s	243296×1	3m 30s	0s	15s
$x^4+468*x^3+79872*x^2+77376*x+82016$	56	62304199776640759616	16619	15s	303420×1	4m 17s	0s	19s
$x^4+1193424*x^2+836604$	432	14549546414559201878 8785600	29032	32s	1852065×1	7m 14s	0s	36s
$x^4+5123374*x^2+20195483$	1	22263620670975556217 4791671076523008	52995	40s	4194303×1	15m 41s	0s	2m 22s
$x^4+52339966*x^2+31445819$	1	37758662123630062538 81932853446926617600	66436	50s	8388607×1	24m 22s	0s	4m 1s
$x^4+514641955*x^2+6061091784$	2	42517873339286813394 81008050927556649428 30664	84478	1m 20s	16777215×1	73m 35s	0s	9m 33s
$x^4+134327397239*x^2+49376458106$	1	25721619871182833530 22950210677742063006 82570799590044064	134970	1m 41s	134217727×1	106m 35s	0s	13m 21s
$x^4+2053826215561*x^2+2108010731420$	68	12978656750375462901 49288230246975023749 34817759881577425180	148214	2m 20s	268435455×1	115m 51s	0s	20m 46s

Tabelle A.3: Körper vom Grade 4 mit 0 reellen Einbettungen.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^4+-8*x^3+-12*x^2+-3*x+-3$	1	-2604555	192	0s	288×1	2s	0s	0s
$x^4+-103*x^3+306*x^2+-315*x+198$	3	-2173994418951	6455	6s	45781×1	1m 15s	0s	2s
$x^4+5318*x^3+5400*x^2+5229*x+-767$	1	-12102355722660227821 6403	22599	15s	2244418×1	5m 43s	0s	29s
$x^4+4561*x^3+97229*x^2+79340*x+75297$	1	-37153300488320162715 34180263	32239	21s	2097151×1	10m 54s	0s	60s
$x^4+425326*x^3+1067901*x^2+-615400*x+1083886$	1	-28298136985265616300 58164226357569088	56358	36s	33554431×1	26m 4s	0s	2m 49s
$x^4+9280823*x^3+-22126013*x^2+-18727051*x+-15986116$	10	-17037501753381455222 83189202281076473721 900	75647	1m 3s	268435455×1	39m 17s	0s	5m 14s
$x^4+-189634521*x^3+483897383*x^2+136761470*x+181920766$	1	-53712143616744964099 78596256664557563701 439428378408	113503	1m 13s	33554431×1	148m 45s	0s	9m 7s
$x^4+-4251666325*x^3+-8834252280*x^2+-8402691825*x+-9130085527$	2	-83947862868350025203 90254201590861875557 8934275941795121412	147267	2m 1s	failed			

Tabelle A.4: Körper vom Grade 4 mit 2 reellen Einbettungen.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^4+-13*x^3+17*x^2+16*x+-17$	1	50309145	2515	2s	3452×1	21s	0s	0s
$x^4+142*x^3+-175*x^2+-314*x+193$	3	72745998970688	8150	8s	145947×1	1m 35s	0s	4s
$x^4+253*x^3+4405*x^2+-5385*x+508$	19	62471111696385524	11965	11s	314545×1	2m 42s	0s	11s
$x^4 + -14852 * x^3 + 74761 * x^2 + 46653 * x + -14735$	1	12174899134579418699426629145	33456	23s	1048575×1	11m 21s	0s	1m 35s
$x^4 + 1778 * x^3 + 571102 * x^2 + 1592834 * x + -265667$	4	13705500439255231329445030160	33579	30s	1048575×1	10m 18s	0s	1m 17s
$x^4+-8700297*x^3+-31578063*x^2+10569011*x+16160941$	1	189157148216171573645821300478541555630064429	83152	55s	262143×1	51m 12s	0s	6m 20s
$x^4 + -105111239 * x^3 + -225468960 * x^2 + 181564367 * x + 399832970$	3	712388195491693197802465231919302760095049114489	101227	1m 26s	failed			

Tabelle A.5: Körper vom Grade 4 mit 4 reellen Einbettungen.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^5+7*x^4+x^3+-6*x^2+-2*x+9$	1	1761289160	816	0s	1300×1	14s	0s	0s
$x^5+-100*x^4+-79*x^3+-45*x^2+71*x+-20$	1	56803974403404049	18603	15s	19466×1	7m 26s	0s	19s
$x^5+-3*x^4+496*x^3+-305*x^2+200*x+-119$	3	2896136587803544293	22588	26s	57402×1	9m 3s	0s	27s
$x^5 + -9398 * x^4 + 4790 * x^3 + -9372 * x^2 + 1749 * x + -3591$	21	40419827300064457438 5814546461	58092	1m 3s	65535×1	38m 54s	0s	2m 30s
$x^5+-69154*x^4+69985*x^3+31354*x^2+-34111*x+-89471$	1	26484504507460776587 67879053769714039705 09	113716	1m 28s	131071×1	114m 0s	0s	10m 20s
$x^5+745606*x^4+-358773*x^3+348169*x^2+-231456*x+528048$	72	13410539096458209256 58833364483852019763 098241	134965	2m 37s	262143×1	183m 39s	0s	13m 5s
$x^5+9001703*x^4+7084557*x^3+-4888154*x^2+-8715456*x+7613945$	6	15806256094325049090 35768781648029490661 67310998489918364	209312	3m 35s	failed			

Tabelle A.6: Körper vom Grade 5 mit einer reellen Einbettung.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^5 + -8 * x^4 + 10 * x^3 + -3 * x^2 + -5 * x + 3$	1	-13709272	226	0s	199×1	4s	0s	0s
$x^5 + -5 * x^4 + 12 * x^3 + -86 * x^2 + 98 * x + 48$	2	-4887973269452	10670	13s	25620×1	4m 31s	0s	6s
$x^5 + -172 * x^4 + 397 * x^3 + -352 * x^2 + -781 * x + -74$	1	-82833454393052586851 20	31838	25s	failed			
$x^5 + 3514 * x^4 + 3223 * x^3 + -7255 * x^2 + 8758 * x + -5243$	1	-33058051844809363744 748223431708	65840	52s	failed			
$x^5 + -42481 * x^4 + 7016 * x^3 + 91450 * x^2 + 51585 * x + -1304$	1	-18572999925176394852 5088131362833465155	97058	1m 16s	131071×1	78m 8s	0s	8m 29s
$x^5 + 302496 * x^4 + -963339 * x^3 + 25373 * x^2 + 130481 * x + 932150$	1	-10728466237582261339 480753739363287449713 39454559	152889	1m 58s	67108863×1	208m 59s	0s	18m 46s
$x^5 + -1605105 * x^4 + 9758939 * x^3 + -3371398 * x^2 + 5039312 * x + -4137956$	2	-18281811458521738209 27975568730701441398 136373333623292	195134	3m 20s	failed			

Tabelle A.7: Körper vom Grade 5 mit 3 reellen Einbettungen.

f	i	Δ	$p_{\max}$	T_f	$a \times b$	T_1	T_2	T_3
$x^5+9*x^4+-11*x^3+-11*x^2+2*x+1$	3	29638353	1045	1s	533×1	16s	0s	0s
$x^5+-19*x^4+-91*x^3+-48*x^2+92*x+48$	8	1185290758525	9661	11s	10498×1	3m 39s	0s	6s
$x^5+280*x^4+984*x^3+-852*x^2+-1001*x+-189$	1	67947729407164150734085	34549	28s	537133×1	17m 22s	0s	60s
$x^5+-2702*x^4+-4398*x^3+7393*x^2+5931*x+-4601$	2	54253744666326928393004083848	58595	1m 3s	65535×1	32m 43s	0s	3m 32s
$x^5+-9081*x^4+54655*x^3+45104*x^2+-42471*x+-4693$	3	11548060245112908639121391941382533 05	86189	1m 31s	131071×1	60m 58s	0s	6m 8s
$x^5+-286951*x^4+-945918*x^3+560736*x^2+844491*x+86318$	3	17779076405750139414627700540347800059352255613	141763	2m 29s	262143×1	173m 27s	0s	29m 44s
$x^5+1981549*x^4+838130*x^3+-8371778*x^2+3532542*x+1190742$	1	22092029435871097614249662672402906854635005775250790 4	188589	2m 26s	268435455×1	349m 34s	0s	50m 39s
$x^5+13078368*x^4+64487754*x^3+-88063411*x^2+-37475037*x+21189636$	15	642288419823925021047448243286243555869022441566382677114 85	237058	4m 11s	failed			

Tabelle A.8: Körper vom Grade 5 mit 5 reellen Einbettungen.

f	i	Δ	$p_{\max}$	T_f	$a \times b$	T_1	T_2	T_3
$x^6+4*x^5+3*x^4+-5*x^3+4*x+4$	2	-292599532	51	0s	162×1	1s	0s	0s
$x^6+-5*x^5+38*x^4+-5*x^3+$	1	-80348267265341427	27273	26s	20177×1	23m 36s	0s	33s
$-30*x^2+39*x+42$								
$x^6+16*x^5+294*x^4+-103*$	1	-15901976672447519339	60611	56s	99653×1	70m 49s	0s	3m 19s
$x^3+-115*x^2+21*x+308$		813092						
$x^6+-72*x^5+1483*x^4+-1769*$	3	-700554511109998167 28	85008	1m 51s	131071×1	111m 27s	7m 37s	5m 4s
$x^3+-144*x^2+1317*x+1881$		5222020355						
$x^6+114*x^5+3271*x^4+-3317*$	1	-42956329594450163246	107953	1m 39s	524287×1	152m 35s	145m 30s	8m 50s
$x^3+-226*x^2+-4247*x+6638$		611820356329 56						
$x^6+347*x^5+101678*x^4+$	1	-607188589242326500 27	200040	3m 6s	2097151×1	438m 42s	failed	
$-60480*x^3+63920*x^2+-4368*$		749841504882895496 59						
$x+22553$		213627						

Tabelle A.9: Körper vom Grade 6 mit 0 reellen Einbettungen.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^6 + -2 * x^5 + 2 * x^4 + -8 * x^3 + -5 * x^2 + 3 * x + 3$	1	4230683577	610	0s	726×1	16s	0s	0s
$x^6 + -29 * x^5 + 42 * x^4 + 30 * x^3 + -36 * x^2 + -22 * x + -18$	1	260893489382578384	28948	27s	failed			
$x^6 + 304 * x^5 + -104 * x^4 + 131 * x^3 + -318 * x^2 + 315 * x + 172$	1	18788487142979523647655688377	76291	1m 9s	65535×1	110m 9s	4m 26s	4m 58s
$x^6 + 1280 * x^5 + 979 * x^4 + 1699 * x^3 + -1514 * x^2 + 351 * x + 148$	2	387795564877443823984114705440888	101352	2m 9s	2097151×1	140m 50s	98m 24s	7m 6s
$x^6 + 14271 * x^5 + -8182 * x^4 + 14585 * x^3 + -6171 * x^2 + -1846 * x + 3105$	1	577382596305790329325447093474761859620415101	174505	2m 37s	131071×1	406m 48s	failed	

Tabelle A.10: Körper vom Grade 6 mit 2 reellen Einbettungen.

f	i	Δ	$p$max	T_f	$a \times b$	T_1	T_2	T_3
$x^6 + -6 * x^5 + 5 * x^4 + 6 * x^3 + 6 * x + -7$	1	-128303422400	10557	10s	7522×1	5m 8s	0s	6s
$x^6 + -35 * x^5 + -3 * x^4 + 4 * x^3 + -38 * x^2 + 38 * x + 32$	22	-20958865744785148	25422	3m 7s	17531×1	19m 2s	0s	32s
$x^6 + 95 * x^5 + -309 * x^4 + -61 * x^3 + 136 * x^2 + 110 * x + -27$	1	-73844773231949123232 5135	54368	49s	320727×1	52m 19s	2m 2s	2m 0s
$x^6 + 117 * x^5 + 1169 * x^4 + -1215 * x^3 + 459 * x^2 + -1763 * x + 1120$	1	-23689742614202016239 54499960464	88048	1m 19s	failed			

Tabelle A.11: Körper vom Grade 6 mit 4 reellen Einbettungen.

f	i	Δ	$p\max$	T_f	$a \times b$	T_1	T_2	T_3
$x^6 + -44 * x^4 + 20 * x^3 + 38 * x^2 + -19 * x + -1$	1	224316530545381	19654	18s	20954×1	9m 31s	0s	21s
$x^6 + 24 * x^5 + -40 * x^4 + -44 * x^3 + 42 * x^2 + 18 * x + -6$	1	11129230881142848	24573	23s	16380×1	16m 3s	0s	30s
$x^6 + 36 * x^5 + 5 * x^4 + -259 * x^3 + -30 * x^2 + 317 * x + -108$	1	255829393923048634807 72	43737	40s	51891×1	42m 45s	0s	1m 36s
$x^6 + 57 * x^5 + -1144 * x^4 + 2012 * x^3 + 375 * x^2 + -1744 * x + 260$	4	45444945894540344035 017389	62824	1m 22s	107066×1	89m 26s	0s	2m 22s
$x^6 + 803 * x^5 + 5471 * x^4 + 6113 * x^3 + -14048 * x^2 + -9191 * x + 263$	1	16651114405174563197 19539306170419161	125208	1m 52s	2097151×1	168m 52s	304m 25s	19m 4s
$x^6 + 9613 * x^5 + -2893 * x^4 + -89672 * x^3 + -30251 * x^2 + 35243 * x + 13321$	1	67279122155154176787 15480909475476159963 348368	200430	3m 0s	262143×1	532m 4s	failed	

Tabelle A.12: Körper vom Grade 6 mit 6 reellen Einbettungen.

Anhang B

Laufzeiten für die HNF-Berechnungen, Ergebnisse

In den folgenden Tabellen ist jeweils f das Körperpolynom, T_H die Zeit zur Berechnung des ersten Teils der HNF, lin die Zahl der linear abhängigen Zeilen in der Relationenmatrix, T_r die Zeit für das Erreichen des vollen Zeilenranges, T_m die Zeit für den zweiten (modularen) Teil der HNF–Berechnung, R der Regulator, sofern er ermittelt werden konnte, h die Klassenzahl als Produkt der Größen der zyklischen Untergruppen und T_C die Zeit zum Suchen zusätzlicher Relationen, falls der Vergleich mit der analytischen Approximation ergibt, dass $h \cdot R$ noch zu groß ist.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^3 + -46 * x^2 + -5 * x + -4$	0.1s	1	0s	0s	22.5978	[2,12]	0s
$x^3 + 180 * x^2 + -664 * x + 1587$	1s	48	45s	0s	99502.9587	[1]	0s
$x^3 + -52240 * x^2 + 57907 * x + -39659$	3.4s	115	2m 11s	2m 24s	80705784.5313	[2,2]	0s
$x^3 + -2112423 * x^2 + 3622665 * x + -4396457$	6m 25s	70	2m 26s	9m 8s	74709614643.5282	[24]	0s
$x^3 + 158148413 * x^2 + 34308993 * x + 186251382$	19m 7s	87	4m 40s	20m 40s	0	[2]	0s
$x^3 + 4133951584 * x^2 + 6810847149 * x + 8211876627$	3h 38m	57	40m 20s	36 m	0	[2]	0s

Tabelle B.1: Körper vom Grade 3 mit einer reellen Einbettung.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^3 + -44 * x^2 + 21 * x + 35$	15s	14	5s	0s	452.5149	[1]	0s
$x^3 + 246 * x^2 + -1377 * x + -1210$	0.3s	21	6s	1s	24328.2208	[2]	0s
$x^3 + -10250 * x^2 + 50113 * x + 41948$	3.1s	73	1m	50s	27017592.9276	[2]	0s
$x^3 + -3318573 * x^2 + -3824415 * x + 1448222$	5m 10s	60	2m 9s	7m 45s	0	[4]	0s
$x^3 + 7406508 * x^2 + -145225378 * x + -131571154$	14m	89	4m 50s	21m 15s	29691017454613 66.4217	[2,2]	0s
$x^3 + 3822941529 * x^2 + 4286379226 * x + 240688561$	1h 11m	90	19m 15s	39m 40s	0	[1]	0s
$x^3 + 88361718083 * x^2 + -352726420334 * x + 311810143202$	6h 56m	115	8h 12m	47m	0	[1]	0s

Tabelle B.2: Körper vom Grade 3 mit 3 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^4 + 245 * x^2 + 266 * x + 191$	5m	105	3m 30s	0s	10511.2136	[2]	0s

Tabelle B.3: Körper vom Grade 4 mit 0 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^4 + -8 * x^3 + -12 * x^2 + -3 * x + -3$	0s	1	0s	0s	172.47	[1]	0s
$x^4 + -103 * x^3 + 306 * x^2 + -315 * x + 198$	4m 44s	106	5m 40s	0s	24342.5097	[2,2,4]	0s
$x^4 + 5318 * x^3 + 5400 * x^2 + 5229 * x + -767$	29s	261	32m 52s	20m 26s	13416439959.2460	[1]	0s
$x^4 + 4561 * x^3 + 97229 * x^2 + 79340 * x + 75297$	51s	326	57m 36s	50m 51s	2948688876236.8156	[3]	0s
$x^4 + 425326 * x^3 + 1067901 * x^2 + -615400 * x + 1083886$	9m 10s	401	13h 13m	1h 35m	166306135883827331.34	[2,2]	0s
$x^4 + 9280823 * x^3 + -22126013 * x^2 + -18727051 * x + -15986116$	30m 46s	432	54h 35m	4h 2m	0	[2]	0s

Tabelle B.4: Körper vom Grade 4 mit 2 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^4+-13*x^3+17*x^2+16*x+-17$	0s	70	1m 9s	0s	523.6606	[1]	0s
$x^4+142*x^3+-175*x^2+-314*x+193$	6m 40s	118	6m 23s	0s	503571.1373	[2]	0s
$x^4+253*x^3+4405*x^2+-5385*x+508$	3s	172	15m 34s	3m 25s	32105098.2802	[1]	0s
$x^4+-14852*x^3+74761*x^2+46653*x+-14735$	1m 19s	355	1h 19m	18m	21038257736910.690151	[1]	0s
$x^4+1778*x^3+571102*x^2+1592834*x+-265667$	37s	408	1h 6m	0s	2232487406921.590771	[1]	0s
$x^4+-8700297*x^3+-31578063*x^2+10569011*x+16160941$	48m 22s	371	61h 27m	9h 57m	0	[1]	0s

Tabelle B.5: Körper vom Grade 4 mit 4 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^5+7*x^4+x^3+-6*x^2+-2*x+9$	0s	29	24s	0s	568.5765	[2]	0s
$x^5+-100*x^4+-79*x^3+-45*x^2+71*x+-20$	2.5s	415	1h 37m	1h 30m	8706971.7312226828787	[1]	0s
$x^5+-3*x^4+496*x^3+-305*x^2+200*x+-119$	5.7s	486	1h 56m	5m 28s	24315868.733650712826	[2,2]	0s
$x^5+-9398*x^4+4790*x^3+-9372*x^2+1749*x+-3591$	2m 6s	742	9h 4m	2h 1m	1268162303450.9783824	[21]	0s

Tabelle B.6: Körper vom Grade 5 mit einer reellen Einbettung.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^5+-8*x^4+10*x^3+-3*x^2+-5*x+3$	0s	7	2s	0s	179.46912707543869287	[1]	0s
$x^5+-5*x^4+12*x^3+-86*x^2+98*x+48$	1.1s	279	1h 4m	0s	530047.2584	[2]	0s
$x^5+-42481*x^4+7016*x^3+91450*x^2+51585*x+-1304$	4m 50s	1191	56h 59m	5h 5m	0	[1]	0s

Tabelle B.7: Körper vom Grade 5 mit 3 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^5+9*x^4+-11*x^3+-11*x^2+2*x+1$	0s	37	19s	0s	203.1239	[1]	0s
$x^5+-19*x^4+-91*x^3+-48*x^2+92*x+48$	0.7s	254	43m	1m 5s	573826.33939892377667	[1]	0s
$x^5+280*x^4+984*x^3+-852*x^2+-1001*x+-189$	16s	628	3h 35m	13m	28591738693.834639442	[1]	0s
$x^5+-2702*x^4+-4398*x^3+7393*x^2+5931*x+-4601$	52s	887	10h 6m	2h 23m	0	[1]	0s
$x^5+-9081*x^4+54655*x^3+45104*x^2+-42471*x+-4693$	2m 57s	1101	31h 44m	5h 37m	0	[2]	0s

Tabelle B.8: Körper vom Grade 5 mit 5 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^6+4*x^5+3*x^4+-5*x^3+4*x+4$	0s	2	4s	0s	84.72200301735579724	[1]	0s

Tabelle B.9: Körper vom Grade 6 mit 0 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^6+-2*x^5+2*x^4+-8*x^3+-5*$ $x^2+3*x+3$	1s	27	1m 3s	0s	696.4438	[1]	0s
$x^6+304*x^5+-104*x^4+131*x^3+$ $-318*x^2+315*x+172$	1m 32s	580	13h 5m	2h 26m	2192715780221.3694209	[1]	0s

Tabelle B.10: Körper vom Grade 6 mit 2 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^6+-6*x^5+5*x^4+6*x^3+6*x+-7$	0s	179	26m 12s	0s	3811.5626301662260215	[1]	0s
$x^6+-35*x^5+-3*x^4+4*x^3+$ $-38*x^2+38*x+32$	3.5s	375	3h 4m	5m	0	[6]	0s
$x^6+95*x^5+-309*x^4+-61*x^3+$ $136*x^2+110*x+-27$	33s	469	5h 1m	38m	0	[1]	0s

Tabelle B.11: Körper vom Grade 6 mit 4 reellen Einbettungen.

f	T_H	lin	T_r	T_m	R	h	T_C
$x^6 + -44 * x^4 + 20 * x^3 + 38 * x^2 + -19 * x + -1$	1.6s	310	1h 2m	0s	0	[1]	0s
$x^6 + 24 * x^5 + -40 * x^4 + -44 * x^3 + 42 * x^2 + 18 * x + -6$	3.2s	346	1h 53	3m	0	[2]	0s
$x^6 + 36 * x^5 + 5 * x^4 + -259 * x^3 + -30 * x^2 + 317 * x + -108$	19.4s	403	5h 45m	2h 39m	0	[1]	0s
$x^6 + 57 * x^5 + -1144 * x^4 + 2012 * x^3 + 375 * x^2 + -1744 * x + 260$	40s	587	12h 5m	21 m	0	[2]	0s
$x^6 + 803 * x^5 + 5471 * x^4 + 6113 * x^3 + -14048 * x^2 + -9191 * x + 263$	5m 1s	875	55h 59m	6h 28m	0	[1]	0s

Tabelle B.12: Körper vom Grade 6 mit 6 reellen Einbettungen.

Anhang C

Ein Beispiel zur Berechnung der Klassengruppe

Hier geben wir den Ablauf der Berechnung für den Körper, der zum Polynom $f(x) = x^4 + 245 * x^2 + 266 * x + 191$ gehört, wieder. Dabei kürzen wir die generierten Ausgaben in dreierlei Hinsicht: wir geben für die Einbettungen weniger Nachkommastellen an, als das Programm ausgibt, wir löschen die Ausgaben des Siebtools und wir löschen die Mitteilungen, welche Relation in welcher Spalte der Relationenmatrix abgespeichert wird weitgehend. Darüber hinaus nehmen wir an durch `(...)` bezeichneten Stellen weitere Kürzungen vor.

```
Prime ideals over primes up to 5991 are
  assumed to generate class group.
Computing in
(x^4 + 245 * x^2 + 266 * x + 191) * 1/3 *
( 3 0 0 2 )
( 0 3 0 0 )
( 0 0 3 2 )
( 0 0 0 1 )
Representation of x with respect to the
  base of O is [ 0 1 0 0 ]
Create factor base.
Bound: Need all non-inert prime ideals over primes up to 5991
Computed factor base in 5 sec 84 hsec
Set number of rows to 854
Sieving on [-39436, 39436] x [1, 2048]
```

```
 At the moment, relations is a 854 x 1 matrix.
Now set it to 1024
COLUMNS is 62
```

Das heißt, wir haben 62 Relationen durch die Zerlegung von Primzahlen in Primideale gefunden, wobei in unserer Faktorbasis 864 Primideale enthalten sind und wir Speicher für 1024 Relationen vorgesehen haben. Nun berechnen wir eine Ganzheitsbasis $(\omega_1, ...\omega_4)$ der Maximalordnung, so dass das Gitter der komplexen Einbettungen LLL-reduziert ist. Für diese algebraischen Zahlen mit (hoffentlich) kleiner Norm berechnen wir dann Polynome der Form $f(x, y) = \mathcal{N}(x + y\omega_i), \quad i = 2, 3, 4$ und testen das Verhalten des Siebtools auf einem kleinen Testintervall, um abzuschätzen, wieviele Relationen wir mit welchem Polynom finden wollen.

```
LLL-reduced base for

( 1.41 -0.76   -0.26     1.06 )
( 0     0.98   -1.06    -0.57 )
( 1.41  0.76 -346.21  -418.25 )
( 0    22.14   24.07 -1786.37 )

is

( 1.41 -0.76 -26.36  146.67 )
( 0     0.98  81.05   79.01 )
( 1.41  0.76  26.36 -148.08 )
( 0    22.14   5.09    7.03 )

Omitting column 0 since it is ==1
Norms are:
        1
        191
Content of N(a+b w_1) = 1
Sieving with primes up to 5981 (782)
Sieving on [-197, 197]x[1, 1]
Found 106 relations for 1
Norms are:
```

```
        1
        1310069
Content of N(a+b w_2) = 1
Sieving with primes up to 5981 (782)
Sieving on [-197, 197]x[1, 1]
Found 94 relations for 2
Norms are:
        1
        152502509
Content of N(a+b w_3) = 1
Sieving with primes up to 5981 (782)
Sieving on [-197, 197]x[1, 1]
Found 79 relations for 3
Computing 0 relations for 0
Computing 300 relations for 1
Computing 266 relations for 2
Computing 224 relations for 3
Computing 300 relations for 1
Norms are:
        1
        191
Content of N(a+b w_1) = 1
Trying to sieve using the polynomial
  x^4 + 245 * x^2 + -266 * x + 191
  generated by [ 1 0 0 0 ] and [ 0 1 0 0 ]
Starting cgr_sieve::sieve(62, 362,1024/1024)
Sieving with primes up to 5981 (782)
Initial sieve array size is [-2000, 2000]
Sieving on [-8000, 8000]x[1, 1]
Found 271 hits.
Sieving on [-8000, 8000]x[2, 2]
Found 398 hits.
Computing 266 relations for 2
Norms are:
        1
        1310069
```

```
Content of N(a+b w_2) = 1
Trying to sieve using the polynomial
  x^4 + 2603 * x^2 + 122018 * x + 1310069
   generated by [ 1 0 0 0 ] and [ 25 82 -1 1 ]
Starting cgr_sieve::sieve(460, 726,1024/1024)
Sieving with primes up to 5981 (782)
Initial sieve array size is [-2000, 2000]
Sieving on [-16000, 16000]x[1, 1]
Found 303 hits.
Computing 224 relations for 3
Norms are:
        1
        152502509
Content of N(a+b w_3) = 1
Trying to sieve using the polynomial
  x^4 + -2 * x^3 + -18573 * x^2 + -626780 * x + 152502509
  generated by [ 1 0 0 0 ] and [ 147 81 0 1 ]
Starting cgr_sieve::sieve(763, 987,1024/1024)
Sieving with primes up to 5981 (782)
Initial sieve array size is [-2000, 2000]
Sieving on [-32000, 32000]x[1, 1]
Found 292 hits.
```

Nachdem wir nun genügend Relationen gefunden haben, versuchen wir zunächst, die Matrix in trivialer Weise zu verkleinern, dazu eliminieren wir einerseits Zeilen und Spalten mit einem Eintrag, wenn dieser 1 oder -1 ist. Darüber hinaus eliminieren wir Nullzeilen, für die Norm des korrespondierenden Erzeugers über der Normschranke für die Erzeuger der Klassengruppe liegt. Dabei wird das Verfahren so lange iteriert, bis keine Änderungen mehr eintreten. Die einzelnen Schritte belassen wir in der Ausgabe, einige Details kürzen wir jedoch.

```
Simplify matrix -- Marking superfluous rows:
854 x 1024 matrix is reduced to a
Removing column 378 with remove_columns!
Removing column 127 with remove_columns!
Remove 192 rows and columns:
(...)
```

```
Removed zero row corresponding to <127, [ 27 40 1 0 ]>
  (norm is too large!).
Removed zero row corresponding to <157, [ 42 44 1 0 ]>
  (norm is too large!).
(...)
Remove 47 rows and columns:
(...)
Remove 10 rows and columns:
(...)
Remove 4 rows and columns:
(...)
Remove 2 rows and columns:
(...)
Remove 5 rows and columns:
(...)
Remove 0 rows and columns:
594 x 766 matrix in time 319 real       319 user        0 sys.
NEEDED: 7458 real       7451 user       7 sys
```

Nun berechnen wir die Hermite-Normalform der Relationenmatrix. Dabei stellen wir fest, dass wir 105 linear abhängige Zeilen in den noch verbliebenen 594 Zeilen haben. Für diese Zeilen generieren wir nun gezielt Relationen, um den vollen Zeilenrang zu erreichen.

```
factorbase: 594 orig_base:  854 removed:    260
Computing 105 additional relations
Calling rel_gen(593,766,...,871)
Generate relations for <5981, [ 4978 1 0 0 ]>
( missing ideal has number 853)
 N1 off by 4055
 N2 off by 2729
(still off by factor 1)
gen_form
Polynomial has content 1
Testing the polynomial
  2729 * x^4 + 332 * x^3 + 7228 * x^2 + -2428 * x + 4055
  generated by [ -1387 -3663 43 -41 ] and [ 4055 0 0 0 ]
```

```
Trying the polynomial
  2729 * x^4 + 332 * x^3 + 7228 * x^2 + -2428 * x + 4055
Starting cgr_sieve::sieve(766, 766,871/976)
Sieving with primes up to 5981 (782)
Sieve array set to an unreasonable value !!!
Initial sieve array size is [-2147483647, 2147483647]
Sieving on [-65535, 65535]x[1, 1]
Found 68 hits.
Store relation to column 766/976
(...)
Store relation to column 833/976
Needed 781 real 778 user        3 sys to find relation for 593
```

Die selbe Prozedur wird nun für die weiteren 104 abhängigen Zeilen durchlaufen, diesen Teil schneiden wir weg, es sei allerdings darauf hingeweisen, dass wir im allgemeinen nicht immer gleich mit dem ersten Polynom eine Relation finden. Allerdings sind die Ausnahmen bei diesem kleinen Beispiel selten. Am schwierigsten war es in diesem Fall, eine Relation für Zeile 450 zu finden, diese Zeilen des Logs wollen wir daher wiedergeben.

```
Calling rel_gen(450,942,...,943)
Generate relations for <3527, [ 244 1 0 0 ]>
( missing ideal has number 570)
 N1 off by 2084
 N2 off by 39532
(still off by factor 4)
gen_form
Polynomial has content 4
Testing the polynomial
  2084 * x^4 + -1792 * x^3 + 17796 * x^2 + -17440 * x + 39532
  generated by [ -1042 0 0 0 ] and [ -2396 -3848 34 -46 ]
Trying the polynomial
   2084 * x^4 + -1792 * x^3 + 17796 * x^2 + -17440 * x + 39532
Starting cgr_sieve::sieve(942, 942,943/976)
Sieving with primes up to 5981 (782)
Sieve array set to an unreasonable value !!!
Initial sieve array size is [-2147483647, 2147483647]
```

```
Sieving on [-65535, 65535]x[1, 1]
Found 0 hits.
 N1 off by 36
 N2 off by 76
(still off by factor 4)
gen_form
Polynomial has content 4
Testing the polynomial
  36 * x^4 + 48 * x^3 + 1004 * x^2 + 304 * x + 76
  generated by [ 6 0 0 0 ] and [ 2 2 0 0 ]
Trying the polynomial
  36 * x^4 + 48 * x^3 + 1004 * x^2 + 304 * x + 76
Starting cgr_sieve::sieve(942, 942,943/976)
Sieving with primes up to 5981 (782)
Initial sieve array size is [-1961849600, 1961849600]
Sieving on [-119741, 119741]x[1, 1]
Found 0 hits.
 N1 off by 21068
 N2 off by 7301
(still off by factor 1)
gen_form
Polynomial has content 1
Testing the polynomial
  7301 * x^4 + -23744 * x^3 + 48262 * x^2 + -9148 * x + 21068
  generated by [ 5171 4547 -49 68 ] and [ -10534 0 0 0 ]
Trying the polynomial
  7301 * x^4 + -23744 * x^3 + 48262 * x^2 + -9148 * x + 21068
Starting cgr_sieve::sieve(942, 942,943/976)
Sieving with primes up to 5981 (782)
Sieve array set to an unreasonable value !!!
Initial sieve array size is [-2147483647, 2147483647]
Sieving on [-65535, 65535]x[1, 1]
Found 43 hits.
Store relation to column 942/976
Needed 534 real 534 user        0 sys to find relation for 450
```

Schließlich wollen wir darauf hinweisen, dass die linear abhängige Zeile mit dem kleinsten Index Zeile 16 ist, die mit dem zweitkleinsten Index ist aber schon Zeile 235. Dies illustriert, dass es typischerweise die Zeilen sind, die großen Primidealen entsprechen, für die wir lineare Abhängigkeiten finden.

Nun führen wir nochmals eine HNF–Berechnung aus. Dies ist in Anbetracht der Tatsache, dass die Matrix zuvor schon einmal auf Hermite–Normalform reduziert wurde, nun leicht möglich und geschieht typischerweise innerhalb weniger Sekunden. Danach führen wir unseren Vereinfachungsschritt durch, der nun alle Primideale, die nichts zur Klassengruppe beitragen, aus der Faktorbasis entfernt. Damit erhalten wir einen vermutetes Ergebnis für die Klassengruppe, das evtl. noch ein Vielfaches des korrekten Ergebnisses ist.

```
compute hnf
Simplify matrix again:
594 x 976 matrix is reduced to a
Listenlaenge:594        Matrixzeilen:594
storing relation to column 853
(...)
storing relation to column 0
Remove 593 rows and columns:
Removed 593 columns.
Remove 0 rows and columns:
1 x 383 matrix in time 108 real 106 user        2 sys.
Resize to 1
Class number might be 2
The class group is generated by (a subset of):
< <19, [ 18 1 0 0 ]>>
( 2 )

( 2 )
```

Schließlich berechnen wir nun - unter Verwendung der analytischen Klassenzahlformel und des Eulerprodukts - eine Approximation an den Regulator, die wir unter der Voraussetzung berechnen, dass die Klassenzahl die korrekte ist. Dann berechnen wir mit Hilfe der gefundenen Einheiten den Regulator - wobei wir aufhören, sobald wir nahe genug an der Approximation sind, und jeweils weitere Einheiten in die Berechnung mit aufnehmen, solange wir nur ein Vielfa-

ches des Regulators gefunden haben. Haben wir alle Einheiten verbraucht und immer noch nur ein Vielfaches des vermuteten Regulators gefunden, so gehen wir dann zu der Annahme über, dass der Regulator korrekt ist und wir nur ein Vielfaches der richtigen Klassennzahl gefunden haben. In diesem Falle müßten wir weitere Relationen berechnen, in unserem Beispiel erweist sich dies jedoch als unnötig.

Anhang D

Bezeichnungen

Zu Beginn wollen wir wichtige Bezeichnungen, die in dieser Arbeit verwendet werden, zusammenfassen. Dabei untergliedern wir diese Aufstellung in folgende Themenbereiche:

- Zahlmengen und ihre Darstellung
- Vektoren und Matrizen
- Polynome
- Spezielle Zahlen
- Normen

Zahlmengen und ihre Darstellung

$\mathbb{N}/\mathbb{N}_0$	Menge der natürlichen Zahlen ohne/mit Null
$\mathbb{P}$	Menge der Primzahlen
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{Q}$	Menge der rationalen Zahlen
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{R}_{>0}/\mathbb{R}_{\geq 0}$	Menge der positiven reellen Zahlen ohne/mit Null
$\mathbb{C}$	Menge der komplexen Zahlen
$\mathbb{F}_p$	Körper mit p Elementen
$\mathbb{K} = \mathbb{Q}(\rho)$	Von ρ erzeugter algebraischer Zahlkörper
$\mathcal{O}$	Ordnung eines Zahlkörpers $\mathbb{K}$
$\mathcal{O}_{\mathbb{K}}$	Maximale Ordnung, Ganzheitsring von $\mathbb{K}$

$\Omega = (\omega_1, \ldots, \omega_n)$	Basis einer Ordnung $\mathcal{O}$ bzw. (Zeilen–)Vektor mit Zahlen der Basis als Komponenten
$\mathrm{MT}(\Omega) = (w_{ijk})_{1 \le i,j,k \le n}$	Multiplikationstafel bzgl. der Basis Ω
$\mathfrak{a}, \mathfrak{b}$	Moduln oder Ideale in einem algebraischen Zahlkörper
$\mathfrak{p}, \mathfrak{q}$	Primideale in einem algebraischen Zahlkörper

Vektoren, Matrizen, Gitter

M^n	Menge der n–stelligen Vektoren mit Einträgen aus M. Vektoren werden meist durch unterstrichene Kleinbuchstaben notiert.
$\underline{e}_i$	i-ter Einheitsvektor
$\underline{a}^T$	Transponierter Vektor zu $\underline{a}$, macht aus einem Zeilen– einen Spaltenvektor.
$\mathrm{M}^{m \times n}$	Menge der $(m \times n)$–Matrizen mit Einträgen aus M. Matrizen werden meist durch Großbuchstaben notiert.
$\mathrm{GL}(n, M)$	Menge der über M invertierbaren $(n \times n)$–Matrizen
I_n	$(n \times n)$–Einheitsmatrix

Polynome

$\deg(f)$	Grad des Polynoms f
$\mathrm{lc}(f)$	Leitkoeffizient des Polynoms f

Spezielle Zahlen

$\mathrm{ggT}(m, n)$	größter gemeinsamer Teiler von m und n
$\Re(c)$	Realteil einer komplexen Zahl c
$\Im(c)$	Imaginärteil einer komplexen Zahl c
σ_i	Einbettungen eines Zahlkörpers $\mathbb{K}$ in $\mathbb{C}$
r_1	Zahl der reellwertigen Einbettungen in $\mathbb{C}$
$2r_2$	Zahl der echt komplexwertigen Einbettungen in $\mathbb{C}$
$\Delta_{\mathcal{O}}$	Diskriminante der Ordnung $\mathcal{O}$
$\det(A)$	Determinante einer quadratischen Matrix A
$\lvert\mathrm{X}\rvert$	Betrag des betragsgrößten Eintrags in der Matrix oder Multiplikationstafel X

Literaturverzeichnis

[Bac90] Bach, E.: Explicit bounds for primality testing and related problems. **In:** *Math. Comp.* 55 (1990), S. 355–380

[Bac98] Backes, W.: *Berechnung kürzester Gittervektoren*, Universität des Saarlandes, Diplomarbeit, 1998

[BD91] Buchmann, J. ; Düllmann, S.: A probabilistic class group and regulator algorithm and its implementation. **In:** *Computational number theory*, 1991, S. 53–72

[BDS93] Bach, E. ; Discroll, J. ; Shallit, J.: Factor Refinement. **In:** *J. Algorithms* 15 (1993), S. 199–222

[BLJ] Buchmann, J. ; Lenstra Jr., H. W.: Computing maximal orders and decomposing primes in number fields. – Vorabdruck

[BLP93] Buhler, J. P. ; Lenstra, H. W. ; Pomerance, C.: Factoring integers with the number field sieve. **In:** Lenstra, A. K. (Hrsg.) ; Lenstra, H. W. (Hrsg.): *The development of the number field sieve*. Springer, 1993 (Lecture Notes in Mathematics 1554), S. 50–94

[Bre88] Bressoud, David M.: *Factorization and Primality Testing*. Heidelberg : Springer, 1988

[Buc89] Buchmann, J.: A subexponential algorithm for the determination of class groups and regulators of algebraic number fields. **In:** *Séminaire de Théorie des Nombres*. Paris, 1988-89, S. 27–41

[BW97] Buchmann, J. ; Williams, H.: Algorithms for binary quadratic forms and quadratic fields. 1997. – unveröffentlicht

[CK91] Cantor, D. G. ; Kaltofen, E.: On fast multiplication of polynomials over arbitrary rings. **In:** *Acta Inform.* 28 (1991), S. 693–701

[Coh95] Cohen, H.: *A course in computational algebraic number theory*. Heidelberg : Springer, 1995

[DKTJ87] Domich, P. D. ; Kannan, R. ; Trotter Jr., L. E.: Hermite normal form computation using modular determinant arithmetic. **In:** *Mathematics of Operations Research* 12 (1987)

[HM89] Hafner, J. L. ; McCurley, K. S.: A rigorous subexponential algorithm for computation of class groups. **In:** *J. Amer. Math. Soc.* 2 (1989), S. 839–850

[How86] Howell, J.A.: Spans in the Module $(\mathbf{Z}_m)^s$. **In:** *Lin. Mult. Alg.* 19 (1986), S. 67–77

[JJ98] Jacobson Jr., M. J.: *Subexponential Class Group Computation in Quadratic Orders*, Technische Universität Darmstadt, Dissertation, 1998

[Mau00] Maurer, M.: *Regulator approximation and fundamental unit computation for real-quadratic orders*, Technische Universität Darmstadt, Dissertation, 2000

[Nei94] Neis, S.: *Kurze Darstellung von Ordnungen*, Universität des Saarlandes, Diplomarbeit, 1994

[Neu92] Neukirch, J.: *Algebraische Zahlentheorie*. Springer, 1992

[OZ58] O. Zariski, P. S.: *Commutative Algebra*. Princeton : Van Nostrand, 1958

[Pap97] Papanikolaou, T.: *Entwurf und Entwicklung einer objektorientierten Bibliothek für algorithmische Zahlentheorie*, Universität des Saarlandes, Dissertation, 1997

[Pfa98] Pfahler, T.: *Polynomfaktorisierung über endlichen Körpern*, Universität des Saarlandes, Diplomarbeit, 1998

[PZ89] Pohst, M. ; Zassenhaus, H.: *Algorithmic Algebraic Number Theory*. CUP, 1989

[98] LiDIA Group: LiDIA *Manual*. 1.3.1. Technische Universität Darmstadt: Institut für Theoretische Informatik, 1 1998

[Ste77] Stender, H.-J.: Lösbare Gleichungen $ax^n - by^n = c$ und Grundeinheiten für einige algebraische Zahlkörper vom Grade $n = 3, 4, 6$. **In:** *J. reine angew. Math.* 290 (1977), S. 24–62

[The96] Theobald, P.: *Implementierung einer linearen Algebra über* $\mathbb{Z}$, Universität des Saarlandes, Diplomarbeit, 1996

[The00] Theobald, P.: *Ein Framework zur Berechnung der Hermite-Normalform von großen, dünnbesetzten, ganzzahligen Matrizen*, Technische Universität Darmstadt, Dissertation, 2000

[Thi95] Thiel, C.: *On the complexity of some problems in algorithmic algebraic number theory*, Universität des Saarlandes, Dissertation, 1995

[Web93] Weber, D.: *Ein Algorithmus zur Zerlegung von Primzahlen in Primideale*, Universität des Saarlandes, Diplomarbeit, 1993

[Web97] Weber, D.: *On the computation of discrete logarithms in finite prime fiel ds*, Universität des Saarlandes, Dissertation, 1997

[Wet98] Wetzel, G. S.: *Various aspects on lattice reduction algorithms and their applications*, Universität des Saarlandes, Dissertation, 1998

[Zay95] Zayer, J.: *Faktorisieren mit dem Number Field Sieve*, Universität des Saarlandes, Dissertation, 1995

Stichwortverzeichnis

Lebenslauf

24.1.1970	Geburt in Merzig
1976 bis 1980	Besuch der Grundschule der Stadt Wadern in Lockweiler
1980 bis 1989	Besuch des Staatlichen Hochwald-Gymnasiums Wadern
10/1989 bis 7/1994	Studium der Informatik an der Universität des Saarlandes in Saarbrücken
17.5.1991	Vordiplom in Mathematik
16.10.1991	Vordiplom in Informatik
10/1991 bis 7/1993	Tätigkeit als studentische Hilfskraft: Betreuung von Übungen zu den Vorlesungen Analysis I/II (WS 91/92 bzw. SS 92) und Informatik I/II (WS 92/93 bzw. SS 93)
22.7.1994	Diplomprüfung in Informatik
9/1994 bis 3/1998	Stelle als wissenschaftlicher Mitarbeiter von Prof. J. Buchmann, zunächst in Saarbrücken, dann in Darmstadt
Seit 4/1998	Angestellter der Kobil Systems GmbH in Worms