JOHANNES GUTENBERG-UNIVERSITÄT MAINZ

Eine generische Implementierung von modularen *ggT*- Algorithmen

Diplomarbeit

vorgelegt dem Fachbereich Physik, Mathematik und Informatik
im Mai 2008
von

Dominik Hülse

Die Arbeit wurde von
Prof. Dr. Elmar Schömer betreut.

Danksagung

Zunächst möchte ich mich bei Herrn Professor Elmar Schömer für das interessante und ergiebige Thema und für die gute Betreuung der Arbeit bedanken.

Mein besonderer Dank gilt meinem Betreuer Michael Hemmer für die hervorragende Zusammenarbeit und die vielen interessanten fachlichen Diskussionen während des vergangenen Jahres.

Schließlich bin ich meinen Eltern, Rosemarie und Heinz Hülse, mehr als dankbar für ihre große Unterstützung während meines gesamten Studiums.

This work was partially supported by the IST Program of the EU *as a Shared-cost* RTD (FET *Open*) *Project under Contract No* IST-006413
*(*ACS *– Algorithms for Complex Shapes with certified numerics and topology).*

Hilfsmittelerklärung (Non-plagiarism statement)
Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Mainz, den 09. Mai 2008,

(Dominik Hülse)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Einordnung der Arbeit

Die Forschung im Bereich *Computational Geometry* war traditionell auf lineare Objekte konzentriert. Was die Implementierung von geometrischen Algorithmen für solche Objekte betrifft, ist die Bibliothek CGAL (*Computational Geometry Algorithms Library*) auf dem neusten Stand der Technik. Die wichtigsten Ziele dieser Algorithmen, die auch in unseren Implementierungen zu berücksichtigen waren, sind Vollständigkeit, Exaktheit und Effizienz. Die Arbeiten an dieser Bibliothek werden unter anderem vom EU–Projekt ACS (*Algorithms for Complex Shapes with certified numerics and topology*) finanziert.
In letzter Zeit gibt es Bemühungen CGAL auf nichtlineare Objekte zu erweitern, speziell auf Objekte, die durch algebraische Kurven und Flächen definiert sind. Als Folge spielt die Arithmetik mit algebraischen Erweiterungen eine wichtigere Rolle. Da CGAL ursprünglich für lineare Objekte gedacht war, konnten unter anderem deren Zahlentypen diese neuen Herausforderungen nicht bewältigen. Es erschien allerdings sehr kompliziert solche fundamentalen Punkte zu ändern, da CGAL schon eine sehr große Bibliothek mit vielen Nutzern war. Aus diesem Grund beschloß eine Gruppe am MPI in Saarbrücken die Forschungen für nichtlineare Objekte aus CGAL auszugliedern und so wurde im April 2002 das EXACUS–Projekt (*Efficient and Exact Algorithms for Curves and Surfaces*) gegründet.
Ein wichtiges Ziel der beiden Projekte ist es, die Wiederverwendbarkeit des erzeugten Codes zu gewährleisten. Das bedeutet, die Implementierungen sollten

so realisiert werden, dass die verwendeten Datentypen möglichst austauschbar sind, ohne den Code anpassen zu müssen. Aus diesem Grund folgen CGAL und EXACUS dem Paradigma der generischen Programmierung, was einen großen Vorteil gegenüber anderen Geometrie Bibliotheken darstellt. Die renommierte Bibliothek NTL zum Beispiel kann nur mit den eigenen Datentypen arbeiten.
Inzwischen wurden schon einige Kernstücke aus EXACUS in CGAL integriert und auch unsere Algorithmen, die im Rahmen des EXACUS–Projekts implementiert wurden, werden in die CGAL Release 3.4 eingegliedert.

Problemstellung

Die Berechnung des größten gemeinsamen Teilers (*ggT*) für Polynome spielt beim exakten geometrischen Rechnen eine wichtige Rolle. Er kommt unter anderem bei der Berechnung der quadratfreien Zerlegung von Polynomen und beim Vergleich algebraischer Zahlen zum Einsatz. Für viele Untersuchungen in EXACUS, beispielsweise während der Berechnung der Nachbarschaftsgraphen einer Anordnung von Quadriken [15], stellte sich heraus, dass der implementierte *ggT*-Algorithmus für univariate Polynome die Hauptschwachstelle in Bezug auf die Laufzeit ist. In der Literatur ist wohlbekannt, dass *ggT*-Algorithmen, basierend auf modularer Arithmetik, den nichtmodularen Methoden weit überlegen sind. Ungeachtet dieser Fakten waren in der EXACUS Release 1.0 von August 2006 noch immer keine modularen *ggT*-Algorithmen implementiert.

Die nichtmodularen *ggT*-Algorithmen bleiben bei der Berechnung des *ggT* in der Struktur $R[X]$ über der die Polynome definiert sind und wenden im Prinzip den euklidischen Algorithmus an. Der Nachteil dieser Methoden ist das bekannte Problem des Koeffizientenwachstums in der polynomiellen Restesequenz während des euklidischen Algorithmus. Schauen wir uns an einem „harmlos“ erscheinendes Beispiel an, wie gewaltig die Koeffizienten im Verlauf der *ggT*-Berechnung anwachsen können und das, obwohl wir nach jeder Division alle überflüssigen Skalare des Restpolynoms entfernen.

Beispiel 1.1 *Für die beiden Polynome*

$$F_1(x) = 476x^8 + 57x^7 + 332x^6 - 15x^5 - 1188x^4 + 1046x^3 - 844x^2 + 1007x - 506,$$
$$F_2(x) = 560x^7 + 72x^6 - 1221x^5 + 1092x^4 + 136x^3 + 286x^2 - 1219x + 529.$$

ergibt sich folgende Restesequenz r_i:

$$\begin{aligned}
r_1 =& 548156x^6 - 380943x^5 - 518164x^4 + 321568x^3 + 77718x^2 + 219283x \\
& - 200813, \\
r_2 =& -355612040x^5 + 1149225926x^4 - 204984605x^3 - 3264093x^2 \\
& - 1148181452x + 668736799, \\
r_3 =& 1540203567476x^4 - 204120359425x^3 - 717536742923x^2 \\
& - 1363418041542x + 1016014543349, \\
r_4 =& -2106559266286048x^3 + 1832510119869860x^2 \\
& + 581524005111801x - 546586833217293, \\
r_5 =& 1301233039402509556x^2 - 2711563822331806497x \\
& + 1349355642522647741, \\
r_6 =& -28x + 23, \\
r_7 =& 0.
\end{aligned}$$

Der gesuchte ggT *ist demnach* $r_6 = -28x + 23$.

Obwohl die Faktoren des eigentlichen *ggT* sehr klein sind, wachsen die Koeffizienten im Verlauf der Rechnung stark an. Dieses Wachstum verstärkt sich für eine längere polynomielle Restesequenz noch weiter. Wie wir in Abbildung 1.1 sehen, weist der nichtmodulare Algorithmus aus EXACUS für steigenden Polynomgrad und konstanten *ggT*-Grad eine exponentielle Laufzeitentwicklung auf. Der Grund dafür ist, dass die Koeffizienten durch die längere Restesequenz noch stärker anwachsen.
Die sogenannten modularen *ggT*-Algorithmen bieten einen Ansatz, um dieses unerwünschte Koeffizientenwachstum zu verhindern. Die prinzipielle Idee dabei ist,

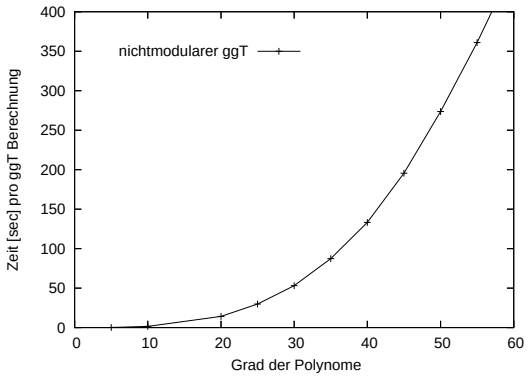
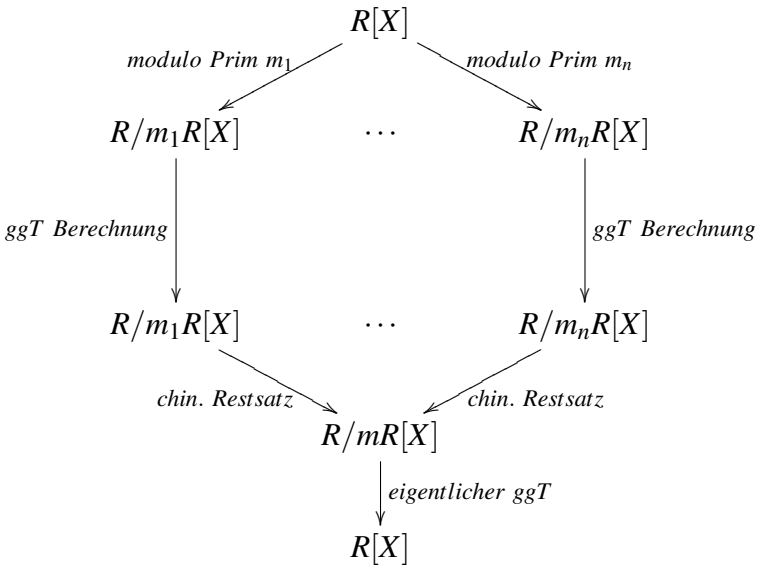


Abbildung 1.1: Zeiten des nichtmodularen Algorithmus für Polynome über $\mathbb{Z}[X]$. Der Grad des ggT ist 1 und die Koeffizientengröße beträgt 5000 Bit.

die Polynome modulo geeigneter Primzahlen in andere Strukturen zu transformieren, in denen der gesuchte ggT effizienter berechnet werden kann. Das Koeffizientenwachstum kann hier nicht auftreten, da die Koeffizientengröße durch die jeweilige Primzahl beschränkt ist. Durch die so gefundenen Lösungen lässt sich mit Hilfe des chinesischen Restsatzes auf die tatsächliche Lösung im ursprünglichen Ring zurückzuschließen.
Das folgende Diagramm fasst das Vorgehen anschaulich zusammen:



Für univariate Polynome über den ganzen Zahlen stellt die Bibliothek NTL einen modularen *ggT*-Algorithmus zur Verfügung. Soweit wir wissen, ist allerdings kein generischer, open source Code verfügbar, der einen modularen *ggT*-Algorithmus für Polynome über algebraischen Erweiterungen unterstützt. Aus diesem Grund, und um EXACUS so weit wie möglich unabängig von anderen Bibliotheken zu halten, haben wir verschiedene modulare *ggT*-Algorithmen für Polynome über den ganzen Zahlen und über algebraischen Erweiterungen vom Grad 2 entwickelt.

Überblick

In Kapitel 2 dieser Arbeit führen wir mathematische Grundlagen ein, um das Problem verstehen und diskutieren zu können. Wir erläutern die Eigenschaften verschiedener Ringe und Erweiterungsringe, definieren den Begriff des *ggT* und führen den euklidischen Algorithmus ein. Danach beschreiben wir das Prinzip der modularen Arithmetik und geben eine für uns wichtige Version des chinesischen Restsatzes an.

Kapitel 3 erläutert ausführlich die von uns implementierten modularen *ggT*-Algorithmen. Zuerst beschreiben wir den Algorithmus von Brown [6] für univariate Polynome über den ganzen Zahlen. Danach gehen wir auf die Schwierigkeiten der *ggT*-Berechnung für Polynome über algebraischen Erweiterungen ein und stellen zwei verschiedene Lösungsansätze vor. Zum einen den Algorithmus von Langemyr und McCallum [22], zum anderen den Algorithmus von Encarnacion [11]. Beide wurden ebenfalls von uns implementiert. Im Anschluss diskutieren wir die Nachteile der beiden Ansätze und entwickeln daraus unseren Hybridansatz. Dieser Ansatz vereinigt die Ideen der beiden Lösungen und erzielte bei unseren Untersuchungen die besten Ergebnisse.

In Kapitel 4 gehen wir auf die Implementierung der Algorithmen, als Teil der EXACUS-Bibliothek, ein. Wir geben zunächst eine kurze Einführung in die Technik der generischen Programmierung, die im EXACUS-Projekt, welches wir danach kurz vorstellen, eine große Rolle spielt. Im Anschluss beschreiben wir das Nullteilerproblem für Polynome über algebraischen Erweiterungen, auf das in der

Literatur keine Antwort zu finden war und präsentieren die von uns entwickelte Lösung. Zum Schluss gehen wir auf Verbesserungen der Implementierung ein, die zu einer Laufzeitoptimierung führten.

Kapitel 5 beschreibt die Laufzeiten unserer Algorithmen für verschiedene Untersuchungsreihen und vergleicht die Ergebnisse mit der bekannten Geometrie Bibliothek NTL und dem Computer Algebra Systemen SINGULAR.

Die Arbeit schließt ab mit einer Zusammenfassung der Ergebnisse und einem Ausblick auf noch offenstehende Probleme in Kapitel 6.

Des Weiteren ist dieser Arbeit eine CD beigelegt, die die Version der EXACUS-Bibliothek zum Zeitpunkt unserer Benchmarks enthält. Die vier Dateien mit unseren Algorithmen, sowie die im Verlauf der Arbeit erwähnten *Traits-Klassen*, sind im Ordner EXACUS/NUMERIX/include/NiX zu finden.
Den Algorithmus von Brown für Polynome über den ganzen Zahlen haben wir `modular_gcd_utcf_algorithm_M.h` genannt. Die Algorithmen für Polynome über algebraischen Erweiterungen sind in den Dateien `modular_gcd_utcf_dfai.h` (Langemyr–McCallum), `modular_gcd_utcf_pure_wang.h` (Encarnacion) und `modular_gcd_utcf_with_wang.h` (Hybridansatz) gespeichert. Außerdem enthält die CD eine elektronische Version dieser Diplomarbeit.

Kapitel 2

Grundlagen

Die Berechnung des größten gemeinsamen Teilers (ggT) ist ein mathematisches Problem aus dem Bereich der Algebra und Zahlentheorie. Um sauber damit arbeiten zu können, definieren wir zuerst ein paar elementare Begriffe, wie sie auch in Büchern über Algebra (z.B. [5], [16] oder [21]) eingeführt werden.

2.1 Ringe und Polynomringe in einer Variablen

Definition 2.1 *Eine Menge R mit Verknüpfungen $+,\cdot : R \times R \to R$ heißt Ring, wenn folgendes gilt:*

1. *$(R,+)$ ist eine abelsche Gruppe (mit dem neutralen Element 0).*

2. *Die Multiplikation $\cdot$ ist assoziativ: Für $a,b,c \in R$ gilt*
   $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

3. *Es gelten die Distributivgesetze*
   $a \cdot (b+c) = a \cdot b + a \cdot c$,
   $(a+b) \cdot c = a \cdot c + b \cdot c$ für $a,b,c \in R$.

Weiterhin gilt:

- *Falls es ein neutrales Element $1 \in R$ bezüglich der Multiplikation gibt, so heißt 1 das Einselement des Ringes und R ein Ring mit Einselement.*

- *Ist die Multiplikation kommutativ, so heißt R ein kommutativer Ring.* [1]
- *Von 0 verschiedene Elemente a und b aus R heißen Nullteiler, wenn $a \cdot b = 0$ gilt. Man nennt $(R, +, \cdot)$ nullteilerfrei, wenn es keine derartigen Elemente gibt.*
- *Ein kommutativer Ring mit Einselement, der nullteilerfrei ist, heißt Integritätsbereich.*
- *Ein kommutativer Ring mit Einselement 1, in dem es zu jedem $a \neq 0$ ein multiplikatives Inverses a^{-1} gibt, heißt Körper.*

Definition 2.2 *Es sei R ein Ring. Eine Teilmenge $\mathfrak{a} \subset R$ heißt ein Ideal in R, wenn gilt:*

1. *$\mathfrak{a}$ ist eine additive Untergruppe von R, d.h. aus $a, b \in \mathfrak{a}$ folgt $a - b \in \mathfrak{a}$.*
2. *Aus $r \in R$, $a \in \mathfrak{a}$ folgt $ra \in \mathfrak{a}$.*

Für ein $a \in R$ nennt man $\langle a \rangle := Ra := \{ra \mid r \in R\}$ das von a erzeugte Ideal.

In unserem Fall ist der Polynomring $R[X]$ über einem kommutativen Ring R in einer Variablen X von besonderer Bedeutung. Anschaulich versteht man unter $R[X]$ die Menge aller Polynome mit Koeffizienten aus einem Ring R und der Variablen X. Der Polynomring $R[X]$ soll nun aber noch auf präzisere Weise konstruiert werden.

Definition 2.3 *Sei R ein kommutativer Ring mit Eins. Wir definieren den Polynomring als den Raum*

$$R[X] := R^{(\mathbb{N}_0)} := \{a = (a_j)_{j \in \mathbb{N}_0} \mid a_j \in R, a_j = 0 \text{ für fast alle } j\}$$

der endlichen Folgen in R. Addition und Multiplikation solcher Folgen seien erklärt durch die Formeln

$$(a_i)_{i \in \mathbb{N}_0} + (b_i)_{i \in \mathbb{N}_0} := (a_i + b_i)_{i \in \mathbb{N}_0}, \tag{2.1}$$

$$(a_i)_{i \in \mathbb{N}_0} \cdot (b_i)_{i \in \mathbb{N}_0} := (c_i)_{i \in \mathbb{N}_0}, \tag{2.2}$$

[1] Wenn nichts anderes gesagt ist, verstehen wir unter einem Ring immer einen kommutativen Ring mit Eins.

wobei

$$c_i := \sum_{i=\mu+\nu} a_\mu b_\nu.$$

Indem man die Ringeigenschaften von R benutzt, kann man leicht nachrechnen, dass $R[X]$ mit diesen Verknüpfungen einen kommutativen Ring mit Eins bildet. Das Nullelement wird gegeben durch die Folge $0 = (0,0,\ldots)$, das Einselement durch die Folge $1 = (1,0,0,\ldots)$. Etwas plausibler wird diese Definition, wenn man für Elemente $f = (a_i)_{i\in\mathbb{N}_0} \in R[X]$ die übliche Polynomschreibweise verwendet:

$$f = (a_i)_{i\in\mathbb{N}_0} = \sum_{i\in\mathbb{N}_0} a_i X^i = \sum_{i=0}^{n} a_i X^i,$$

wobei n so groß gewählt ist, dass $a_i = 0$ für $i > n$ gilt.
Weiter definiert man den *Grad* von f durch

$$Grad(f) := max\{i, a_i \neq 0\};$$

dem Nullpolynom wird der Grad $-\infty$ zugeordnet. Im Fall $Grad(f) = n \geq 0$ heißt a_n der *Leikoeffizient* von f. Ist dieser 1, so sagt man, f sei *normiert*.

Bemerkung 2.4 *Sei $R[X]$ der Polynomring einer Variablen X über einem kommutativen Ring R. Für Polynome $f,g \in R[X]$ gilt dann*

$$Grad(f+g) \leq max(Grad(f), Grad(g))$$
$$Grad(f\cdot g) \leq Grad(f) + Grad(g),$$

wobei man sogar $Grad(f\cdot g) = Grad(f) + Grad(g)$ hat, sofern R ein Integritätsbereich ist.

Bemerkung 2.5 *Ist R ein Integritätsbereich, dann ist auch der Polynomring $R[X]$ ein Integritätbereich.*

Um zwei Polynome auf das Vorhandensein gemeinsamer Nullstellen zu prüfen, wird oft die Resultante der beiden Polynome berechnet.

Definition 2.6 *Seien $f = f_0 + f_1X + \ldots + f_mX^m$, $g = g_0 + g_1X + \ldots + g_nX^n \in R[X]$ zwei Polynome über dem Ring R mit $Grad(f) = m$ und $Grad(g) = n$, so ist die Resultante von f und g definiert durch:*

$$\operatorname{res}(f,g) = \det \begin{pmatrix} f_m & \cdots & f_0 & & & \\ & f_m & & f_0 & & \\ & & \ddots & & \ddots & \\ & & & f_m & \cdots & f_0 \\ g_n & \cdots & g_0 & & & \\ & g_n & & g_0 & & \\ & & \ddots & & \ddots & \\ & & & g_n & \cdots & g_0 \end{pmatrix}.$$

Dabei stehen die Koeffizienten von f in $Grad(g) = n$ Zeilen und die Koeffizienten von g in $Grad(f) = m$ Zeilen. Alle restlichen Einträge der Matrix sind Null.

Die Resultante ist genau dann Null, wenn die beiden Polynome eine gemeinsame Nullstelle haben.

Definition 2.7 *Sei R ein Integritätsbereich und $a, b \in R$.*

- *Wir sagen a teilt b, falls $b = ac$ für ein $c \in R$. Dies schreibt man als $a \mid b$. Nicht-Teilbarkeit wird als $a \nmid b$ geschrieben.*
- *Die Einheiten des Ringes sind Teiler der Eins, $R^\times := \{u \in R : u \mid 1\}$.*
- *Die Elemente a, b heißen assoziiert, falls sie sich nur um eine Einheit unterscheiden, also $a = ub$ für ein $u \in R^\times$.*
- *Oft ist es dienlich, aus einer Menge von assoziierten Elementen einen Repräsentanten auszuwählen und diesen als einheitsnormal zu definieren. Für $a \in R$ bezeichnen wir das einheitsnormale assoziierte Elemente mit $unit\_normal(a)$.*

In $\mathbb{Z}$ sind alle nichtnegativen Zahlen einheitsnormal. Da in einem Körper alle Elemente ungleich Null Einheiten sind, gilt nur die 1 als einheitsnormal. Im Polynomring sind es alle Polynome mit einheitsnormalem Leitkoeffizienten.

Sei weiterhin $p \in R \backslash R^{\times}$

- *p heißt prim, falls für alle $r, s \in R$ aus $p \mid rs$ folgt, dass $p \mid r$ oder $p \mid s$.*
- *p heißt irreduzibel, falls aus $p = rs$ für $r, s \in R$ folgt, dass p zu r oder s assoziiert ist. (Der andere Faktor ist damit eine Einheit.)*
- *p heißt reduzibel, falls p nicht irreduzibel ist.*
- *Ein primes Element ist immer irreduzibel. (Die Umkehrung gilt i.A. nicht.)*

Definition 2.8 *Ein faktorieller Ring ist ein Integritätsbereich, in dem jedes Element, das weder Null noch eine Einheit ist, eindeutig in ein Produkt von Primelementen zerlegt werden kann (Primfaktorzerlegung).*

Lemma 2.9 (Lemma von Gauß) *Ist R ein faktorieller Ring, dann ist $R[X]$ auch ein faktorieller Ring.*

Ein Beweis wird zum Beispiel im Buch von Bosch [5] Kapitel 2.7 präsentiert.

Bemerkung 2.10 *In einem faktoriellen Ring ist ein irreduzibles Element prim.*

Um den euklidischen Algorithmus zur *ggT* Bestimmung benutzen zu können, benötigt man einen Ring, in dem eine Division mit Rest existiert. Diese Ringe nennt man euklidische Ringe.

Definition 2.11 *Ein Integritätsbereich R heißt euklidisch, falls eine Norm-Funktion*

$$N : R \backslash \{0\} \longrightarrow \mathbb{N}_0$$

existiert, so dass gilt:
Sind $a, b \in R$ mit $b \neq 0$, dann gibt es $q, r \in R$ mit $a = qb + r$, wobei entweder $r = 0$ oder $r \neq 0$ und $N(r) < N(b)$ gilt.

Neben den ganzen Zahlen $\mathbb{Z}$ ist der Polynomring $R = K[X]$ in einer Variablen über dem Körper K das bekannteste Beispiel für einen euklidischen Ring. Hier ist $N(f) = grad(f)$ die Norm-Funktion.

2.2 *ggT* und euklidischer Algorithmus

2.2.1 Der größte gemeinsame Teiler

Definition 2.12 *Es sei R ein faktorieller Ring und $a, b \in R$, nicht beide 0. Ein Element $g \in R$ heißt ein größter gemeinsamer Teiler (*ggT*) von a und b, wenn gilt:*

- $g \mid a$ *und* $g \mid b$
- *aus* $t \mid a$ *und* $t \mid b$ *folgt* $t \mid g$
- *g ist einheitsnormal*

Zwei Elemente a und b werden als *teilerfremd* bezeichnet, wenn $ggT(a,b) = 1$ ist. Die folgenden Beziehungen sind elementare Eigenschaften der Funktion *ggT*.

Korollar 2.13 *Seien a, b, c Elemente eines faktoriellen Ringes R.*

$$ggT(a,b) = ggT(b,a), \tag{2.3}$$
$$ggT(a,b) = ggT(-a,b), \tag{2.4}$$
$$ggT(a,b) = ggT(unit\_normal(a), unit\_normal(b)), \tag{2.5}$$
$$ggT(a,0) = unit\_normal(a), \tag{2.6}$$
$$ggT(a,ka) = unit\_normal(a), \text{ für jedes } k \in R, \tag{2.7}$$
$$ggT(a,b) = ggT(a,b+ca), \tag{2.8}$$
$$ggT(ab,c) = ggT(a,c) \cdot ggT(b,c), \text{wenn a und b teilerfremd.} \tag{2.9}$$

2.2.2 Der euklidische Algorithmus

Mit Hilfe des euklidischen Algorithmus läßt sich der *ggT* mit geringerem Aufwand berechnen als mit einer Primfaktorzerlegung. Er beruht auf den Eigenschaften (2.6) und (2.8).

Satz 2.14 (Euklidischer Algorithmus) *Seien r_0 und r_1 Elemente eines euklidischen Ringes. Man berechne r_{i+1} als Rest der Division von r_{i-1} durch r_i, d.h. mit $q_{i+1} = \lfloor r_{i-1}/r_i \rfloor$*[2]

$$r_{i-1} = q_{i+1} r_i + r_{i+1} \quad \textit{mit } N(r_{i+1}) < N(r_i) \textit{ oder } r_{i+1} = 0,$$

bis $r_{k+1} = 0$. Dann ist $ggT(r_0, r_1) = r_k$.

Beweis Wegen $N(r_1) > N(r_2) > N(r_3) > \ldots$ muss nach endlichen vielen Schritten die 0 erreicht werden. Nach Korollar 1.5 ist für $0 \leq i \leq k$

$$ggT(r_{i-1}, r_i) = ggT(q_{i+1} r_i + r_{i+1}, r_i) = ggT(r_{i+1}, r_i) = ggT(r_i, r_{i+1})$$

und daher

$$ggT(r_0, r_1) = ggT(r_1, r_2) = \ldots = ggT(r_k, r_{k+1}) = ggT(r_k, 0) = r_k\,. \qquad \square$$

Beispiel 2.15 *Der* ggT *von 7497 und 5145 wird mit dem euklidischen Algorithmus wie folgt berechnet:*

$$\begin{aligned} 7497 &= 1 \cdot 5145 + 2352 \\ 5145 &= 2 \cdot 2352 + 441 \\ 2352 &= 5 \cdot 441 + 147 \\ 441 &= 3 \cdot 147 + 0 \end{aligned}$$

Es gilt somit $ggT(7497, 5145) = 147$.

Der euklidische Algorithmus für Polynome

Polynome in einer Variable über einem Körper bilden einen euklidischen Ring. Auch hier kann also zur Bestimmung des *ggT* der euklidische Algorithmus angewendet werden.

Beispiel 2.16 *Die Berechnung des* ggT *der Polynome $f = x^5 + 3x^3 + 2x$ und $g = x^3 + x^2 + x + 1$ aus $\mathbb{Q}[x]$ gestaltet sich folgendermaßen:*

$$\begin{aligned} x^5 + 3x^3 + 2x &= (x^2 - x + 3) \cdot (x^3 + x^2 + x + 1) + (-3x^2 - 3) \\ x^3 + x^2 + x + 1 &= (-\frac{1}{3}x - \frac{1}{3}) \cdot (-3x^2 - 3) + 0. \end{aligned}$$

Damit ist $3x^2 + 3$ der ggT *von f und g.*

[2] Für eine reelle Zahl x ist $\lfloor x \rfloor$ die größte ganze Zahl, die kleiner oder gleich x ist.

Definition 2.17 *Wird der euklidische Algorithmus für zwei Polynome $r_0, r_1 \in K[X]$ angewendet, nennen wir die sich ergebende polynomielle Restefolge $(r_0, r_1, r_2, \ldots, r_k)$ PRS (polynomial remainder sequence).*

Wie das Beispiel zeigt, ist der euklidische Algorithmus für Polynome nur anwendbar, wenn der Grundring ein Körper ist. Die *ggT*-Berechnung über $\mathbb{Z}[x]$ würde offensichtlich nicht funktionieren, da der Quotient $(-\frac{1}{3}x - \frac{1}{3})$ nicht in $\mathbb{Z}[x]$ liegt. Deswegen wird eine sogenannte *Pseudo-Division* definiert, die über allen Integritätsbereichen anwendbar ist.

Lemma 2.18 (Pseudo-Division) *Sei R ein Integritätsbereich, f und g Polynome aus $R[x]$ mit Grad d_f bzw. d_g. Sei g_0 der Leitkoeffizient des Polynoms g und $\delta := d_f - d_g$. Dann gibt es Polynome $q, r \in R[x]$, so dass*

$$g_0^{\delta+1} f = qg + r,$$

wobei wieder r von geringerem Grad ist als g.

Durch wiederholte Durchführung der Pseudodivision lässt sich der *ggT* von f und g bestimmen. Allerdings ist dieses Verfahren in der Praxis äußerst ineffizient.

Beispiel 2.19 *Benutzen wir in Beispiel 2.16 die Pseudo-Division, so funktioniert die ggT-Berechnung auch für $f, g \in \mathbb{Z}[x]$.*

$$\begin{aligned} x^5 + 3x^3 + 2x &= (x^2 - x + 3) \cdot (x^3 + x^2 + x + 1) + (-3x^2 - 3) \\ 9 \cdot (x^3 + x^2 + x + 1) &= (-3x - 3) \cdot (-3x^2 - 3) + 0. \end{aligned}$$

2.2.3 Der erweiterte euklidische Algorithmus

Wir formulieren den euklidischen Algorithmus nun so um, dass er zusätzliche Informationen berechnet. Insbesondere wollen wir die Elemente s und t der Darstellung

$$d = ggT(a,b) = as + bt$$

berechnen. Diese Elemente werden später im chinesischen Restsatz benötigt. In unserem Beispiel 2.15 geht das durch die folgende Rückwärtsrechnung, begin-

nend bei der vorletzten Zeile:

$$\begin{aligned}
147 &= 2352 - 5 \cdot 441 \\
&= 2352 - 5 \cdot (5145 - 2 \cdot 2352) \\
&= -5 \cdot 5145 + 11 \cdot 2352 \\
&= -5 \cdot 5145 + 11 \cdot (7497 - 1 \cdot 5145) \\
&= 11 \cdot 7497 - 16 \cdot 5145.
\end{aligned}$$

Jetzt formalisieren wir diese Rechnung.

Satz 2.20 (Erweiterter euklidischer Algorithmus (EEA)) *Seien a und b Elemente eines euklidischen Ringes. Man setze* $r_0 = a,\ r_1 = b,\ s_0 = 1,\ t_0 = 0,\ s_1 = 0$ *und* $t_1 = 1$ *und berechne*
$r_{i+1},\ q_{i+1},\ s_{i+1},\ t_{i+1}$ *für* $i \geq 1$ *wie folgt*

$$\begin{aligned}
q_{i+1} &= \lfloor r_{i-1}/r_i \rfloor \\
r_{i-1} &= q_{i+1} r_i + r_{i+1} \qquad \textit{mit } N(r_{i+1}) < N(r_i) \textit{ oder } r_{i+1} = 0 \\
s_{i+1} &= s_{i-1} - q_{i+1} s_i \\
t_{i+1} &= t_{i-1} - q_{i+1} t_i
\end{aligned}$$

solange bis $r_{k+1} = 0$.
Dann ist $ggT(a,b) = r_k = s_k a + t_k b$.

Beweis Nach Satz 1.6 wissen wir, dass $ggT(a,b) = r_k$ gilt. Es reicht also, mittels Induktion zu zeigen, dass für alle i

$$r_i = s_i a + t_i b$$

gilt. Der Induktionsanfang für $i = 0, 1$ ist durch die Definition der s_0, s_1, t_0, t_1 klar. Der Induktionsschritt $(i \to i+1)$ läuft wie folgt

$$\begin{aligned}
r_{i+1} &= r_{i-1} - q_{i+1} r_i \\
&\overset{I.V.}{=} (s_{i-1} a + t_{i-1} b) - q_{i+1}(s_i a + t_i b) \\
&= (s_{i-1} - q_{i+1} s_i) a + (t_{i-1} - q_{i+1} t_i) b \\
&= s_{i+1} a + t_{i+1} b.
\end{aligned}$$

□

2.3 Modulare Arithmetik und Chinesischer Restsatz

2.3.1 Modulare Arithmetik

In Definition 2.11 haben wir gesehen, dass in euklidischen Ringen eine *Division mit Rest* existiert. Diese benutzen wir nun, um modulare Arithmetik zu definieren. Sei im Folgenden stets R ein euklidischer Ring. (Wir können dabei immer an $R = \mathbb{Z}$ denken.)

Definition 2.21 *Sei $0 \neq m \in R$. Definiere für $f, g \in R$ die Relation*

$$f \equiv g \mod m \ :\Leftrightarrow f = g + km,\ k \in R$$

Gesprochen: f kongruent g modulo m.

Lemma 2.22 *Kongruenz modulo m ist eine Äquivalenzrelation auf R, d.h für $0 \neq m \in R$ und für alle $f, g, h \in R$ gelten:*

1. *(Reflexivität) $f \equiv f \mod m$,*
2. *(Symmetrie) $f \equiv g \mod m \Rightarrow g \equiv f \mod m$,*
3. *(Transitivität) $f \equiv g \mod m$ und $g \equiv h \mod m \Rightarrow f \equiv h \mod m$.*

Bei Äquivalenzrelationen betrachtet man immer auch die Äquivalenzklassen.

Definition 2.23 *Für $m \in R$ heißen die Äquivalenzklassen auch Restklassen mod m und sind konkret definiert durch*

$$\overline{a} := a + mR := \{a + km \mid k \in R\}$$

Damit definiere den Restklassenring von R modulo m

$$R_m := R/mR := \{a + mR \mid a \in R\}$$

und auf R/mR folgende Operationen

$$\overline{a} + \overline{b} := \overline{a + b} \quad \text{und} \quad \overline{a} \cdot \overline{b} := \overline{a \cdot b}.$$

Es existiert ein natürlicher Ringhomomorphismus

$$\phi_m : R \to R/mR \quad \text{mit} \quad \phi_m(x) := x \mod m.$$

Das homomorphe Bild von ϕ_m wird als modulares Bild bezeichnet.

Im Augenblick mag R/mR noch als große Menge erscheinen. Dies ist aber nicht der Fall. Am Beispiel $R = \mathbb{Z}$ versteht man $\mathbb{Z}/m\mathbb{Z}$ am besten über ein vollständiges Repräsentantensystem. Das ist eine Menge $L \subseteq \mathbb{Z}$ mit $\mathbb{Z}/m\mathbb{Z} = \{l + m\mathbb{Z} \mid l \in L\}$, und zwei $l_1, l_2 \in L$ mit $l_1 \neq l_2$ haben verschiedene Restklassen $l_1 + m\mathbb{Z} \neq l_2 + m\mathbb{Z}$.

Lemma 2.24 *$L = \{0, 1, \ldots, m-1\}$ ist ein Repräsentantensystem für $\mathbb{Z}/m\mathbb{Z}$. Insbesondere besteht $\mathbb{Z}/m\mathbb{Z}$ aus m Elementen.*

Satz 2.25 *Mit den Operationen aus 2.23 wird R/mR zu einem kommutativen Ring mit Einselement $\overline{1}$ und Nullelement $\overline{0}$. Das additive Inverse zu $\overline{a}$ ist $-\overline{a}$.*

Beweis Folgt direkt aus Lemma 2.22. □

Beispiel 2.26 *Für $R = \mathbb{Z}$ und $m = 5$ ergeben sich folgende Tafeln:*

$\oplus$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$\odot$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

In der Multiplikationstafel sehen wir, dass jedes $\overline{a}$ ein multiplikativ Inverses besitzt. Es stellen sich also nun folgende Fragen:

- Ist dies immer der Fall?
- Was sind die Bedingungen für die Existenz eines Inversen?

Lemma 2.27 *Sei wieder $0 \neq m \in R$. Dann gilt:*

$$\textit{n ist genau dann ein Nullteiler in } R/mR, \textit{ wenn } ggT(m,n) \neq 1. \tag{2.10}$$

$$\textit{Ist } ggT(f,m) = 1, \textit{ dann existiert ein } \overline{i} \textit{ mit } \overline{i} \cdot \overline{f} = \overline{1}. \tag{2.11}$$

$$\textit{Ist m irreduzibel, so ist } R/mR \textit{ ein Körper.} \tag{2.12}$$

Für $R = \mathbb{Z}$ gilt also:

$$\textit{Ist } p \in \mathbb{Z} \textit{ prim, so ist } \mathbb{Z}_p := \mathbb{F}_p := \mathbb{Z}/p\mathbb{Z} \textit{ ein Körper mit p Elementen.} \tag{2.13}$$

2.3.2 Der chinesische Restsatz

Der chinesische Restsatz ist einer der anwendungsreichsten Sätze der elementaren Zahlentheorie. In unserem Fall ermöglicht er es uns, die Koeffizienten der Polynome aus ihren Bildern modulo $m_1, \ldots, m_n$ zu rekonstruieren. Eine Version des Satzes lautet:

Satz 2.28 (Chinesischer Restsatz) *Seien $m_1, m_2, \ldots, m_n$ paarweise teilerfremde ungerade ganze Zahlen und $m = \prod_{i=1}^{n} m_i$, dann existiert für jedes Tupel ganzer Zahlen $u_1, \ldots, u_n$ eine eindeutige ganze Zahl x mit $0 \leq x < m$, die die folgende simultane Kongruenz erfüllt:*

$$x \equiv u_i \mod m_i \quad \textit{für } i = 1, \ldots, n.$$

Beweis Der hier präsentierte Beweis ist nützlich für unsere Algorithmen und orientiert sich am Vorgehen von Brown [6].
Gegeben seien m_1, m_2 teilerfremde natürliche Zahlen, die sogenannten Moduli und u_1, u_2 zwei ganze Zahlen. Dann folgt mit Hilfe des chinesischen Restsatzes: Es existiert eine eindeutige ganze Zahl u, so dass $u \equiv u_1 \mod m_1$, $u \equiv u_2 \mod m_2$ und $0 \leq u < m_1 m_2$.

Eindeutigkeit: Wir nehmen an es existieren zwei Zahlen u *und* u', die beide die Voraussetzungen erfüllen. Aus $u \equiv u_1 \mod m_1$, $u \equiv u_2 \mod m_2$ und $u' \equiv u_1 \mod m_1$, $u' \equiv u_2 \mod m_2$ folgt mit Lemma 2.22

$$u \equiv u' \mod m_1 \text{ und } u \equiv u' \mod m_2.$$

Nach Definition 2.21 bedeutet das $m_1 \mid (u - u')$ und $m_2 \mid (u - u')$. Da m_1, m_2 teilerfremd, ist das kleinste gemeinsame Vielfache (kgV) von m_1 *und* m_2 gerade $m_1 m_2$. Deshalb folgt $m_1 m_2 \mid (u - u')$. Wegen $0 \leq u, u' < m_1 m_2$ folgt $u = u'$.

Existenz: Da m_1, m_2 teilerfremd, liefert der erweiterte euklidische Algorithmus (Satz 2.20) Zahlen s und t, so dass $sm_1 + tm_2 = 1$. Daraus folgt $sm_1 \equiv 1 \mod m_2$. Setze $v = s(u_2 - u_1) \mod m_2$. Damit gilt $m_1 v = m_1 s(u_2 - u_1) \equiv u_2 - u_1 \mod m_2$. Wenn wir nun $u' = m_1 v + u_1$ setzen und $u \equiv u' \mod m_1 m_2$ mit $0 \leq u < m_1 m_2$ wählen, sind wir fertig.

Da $m_1, m_2, \ldots, m_n$ teilerfremd gewählt, folgt die allgemeine Aussage sukzessive. □

Bemerkung 2.29 *Da wir in unserer Anwendung auch negative Zahlen rekonstruieren möchten, wählen wir nicht* $\{0,1,\ldots,m-1\}$ *als Repräsentantensystem von* $\mathbb{Z}/m\mathbb{Z}$ *sondern* $R = \{-\lfloor m/2 \rfloor, \lfloor m/2 \rfloor\}$. *R ist auch ein vollständiges Repräsentantensystem von* $\mathbb{Z}/m\mathbb{Z}$, *da es genau m Elemente enthält:*

$$\mid R \mid = 2 \cdot \lfloor m/2 \rfloor + \underbrace{1}_{die\ 0} \overset{m\ ungerade}{=} m-1+1 = m$$

2.4 Algebraische Erweiterungen

Will man Polynomgleichungen untersuchen, deren Lösungen noch nicht im Koeffizientenring liegen, etwa quadratische Gleichungen mit ganzzahligen Koeffizienten, so ist es zweckmäßig geeignete Erweiterungsringe des Koeffizientenringes zu betrachten. Wir führen grundlegende Begriffe und Eigenschaften solcher Erweiterungen ein und definieren den Begriff der algebraischen Zahl. Eine vollständige Darstellung ist in standard Algebra Büchern wie Lang [21] oder Hungerford [16] zu finden.

2.4.1 Körpererweiterungen und algebraische Zahlen

Wir gehen zuerst auf den bekannteren Spezialfall der Körpererweiterung ein:

Definition 2.30 *Seien K, L Körper mit* $K \subseteq L$.

1. *L wird Oberkörper von K genannt. Das Paar L und K bezeichnen wir als Körpererweiterung und schreiben es als* L/K *(„L über K“).*

2. *Man kann L als Vektorraum über K auffassen. Die Dimension dieses Vektorraums* $(\dim_K L)$ *nennt man den Grad der Körpererweiterung, und schreibt ihn als* $[L:K]$. *Ist* $[L:K]$ *endlich, so sagt man, die Erweiterung* L/K *sei eine endliche Erweiterung.*

3. *Ein Element* $\alpha \in L$ *heißt algebraisch über K, wenn es ein Polynom* $0 \neq f \in K[X]$ *gibt mit* $f(\alpha) = 0$. *Andernfalls heißt* α *transzendent über K.*

4. L/K *heißt algebraische Erweiterung, wenn alle* $\alpha \in L$ *algebraisch über K sind.*

5. *Ist $\alpha \in L$ algebraisch über K, so heißt das normierte Polynom M von kleinstem Grad in $K[X]$ mit $M(\alpha) = 0$ das Minimalpolynom von α über K.*

Bemerkung 2.31 *Sei $M \in K[X]$ das Minimalpolynom von α und $N \in K[X]$ ein weiteres Polynom mit $N(\alpha) = 0$. Dann sind folgende Forderungen äquivalent*

1. $grad(M) \leq grad(N)$.

2. $M \mid N$.

Beweis **2.** $\Rightarrow$ **1.** trivial.
1. $\Rightarrow$ **2.** Aus $grad(M) \leq grad(N)$ folgt durch Division mit Rest $N = sM + r$ mit $grad(r) < grad(M)$. Weiterhin gilt $0 = N(\alpha) = s(\alpha)M(\alpha) + r(\alpha)$ und damit $r(\alpha) = 0$. Wegen $grad(r) < grad(M)$ folgt $r \equiv 0$ und damit die Behauptung. □

Definition 2.32 *Sei L/K eine Körpererweiterung und $\alpha \in L$ algebraisch über K. Wir bezeichnen mit $K(\alpha)$ den kleinsten Unterkörper von L, welcher K und α enthält. Das heißt $K(\alpha)$ ist der kleinste Körper, der von K und α erzeugt wird.*

Lemma 2.33 *Sei wieder L/K eine Körpererweiterung und $\alpha \in L$ algebraisch über K mit Minimalpolynom $M \in K[X]$. Für den Grad der Körpererweiterung $K(\alpha)/K$ gilt dann*

$$[K(\alpha) : K] = grad(M).$$

Den Beweis findet man zum Beispiel im Buch von Bosch [5] Kapitel 3.2.

Lemma 2.34 *Mit den Voraussetzungen aus Lemma 2.33 gilt*

$$K(\alpha) \simeq K[X]/\langle M \rangle$$

Beweis Definieren wir uns den Einsetzungshomomorphismus $\phi : K[X] \to K(\alpha)$ durch $f \mapsto f(\alpha)$. Aus Bemerkung 2.31 folgt $Kern(\phi) = \langle M \rangle$ und dadurch mit Hilfe des Homomorphiesatzes $K(\alpha) \simeq K[X]/\langle M \rangle$. □

2.4.2 Erweiterungen der rationalen und der ganzen Zahlen

Konzentrieren wir uns nun auf den Körper der rationalen Zahlen. Das Minimalpolynom einer algebraischen Zahl α hat die Gestalt

$$M(X) = X^m + \sum_{i=0}^{m-1} b_i X^i, \quad b_i \in \mathbb{Q} \tag{2.14}$$

Nach Lemma 2.34 gilt $\mathbb{Q}(\alpha) \simeq \mathbb{Q}[X]/\langle M(X)\rangle$. Deshalb läßt sich ein Element $\beta \in \mathbb{Q}(\alpha)$ eindeutig darstellen als

$$\beta = \sum_{i=0}^{m-1} b_i \alpha^i = B(\alpha), \quad b_i \in \mathbb{Q}. \tag{2.15}$$

Beispiel 2.35 *Sei $d \in \mathbb{N}$ kein Quadrat in $\mathbb{N}$. Dann ist d auch kein Quadrat in $\mathbb{Q}$. $\sqrt{d}$ hat über $\mathbb{Q}$ das Minimalpolynom $M = X^2 - d$. Der Grad der Körpererweiterung $\mathbb{Q}(d)/\mathbb{Q}$ ist also 2. Anders gesagt, 1 und $\sqrt{d}$ bilden eine Basis des $\mathbb{Q}$-Vektorraums $\mathbb{Q}(\sqrt{d})$, also gilt*

$$\mathbb{Q}(\sqrt{d}) = \{a + b\sqrt{d} \in \mathbb{R} \mid a, b \in \mathbb{Q}\}.$$

Erweiterungen dieses Typs heißen quadratische Erweiterungen.

Definition 2.36 *Eine Zahl α heißt ganze algebraische Zahl, wenn sie Nullstelle eines normierten Polynoms mit Koeffizienten aus $\mathbb{Z}$ ist.*
Das Polynom kleinsten Grades mit dieser Eigenschaft nennen wir wieder Minimialpolynom von α.
Für α definieren wir den Integritätsring der ganzen algebraischen Erweiterung von $\mathbb{Z}$ als

$$\mathbb{Z}(\alpha) = \{a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1} : a_i \in \mathbb{Z}\}.$$

Wenn wir im Folgenden von einer algebraischen Erweiterung der ganzen Zahlen sprechen, meinen wir immer eine ganze algebraische Erweiterung. Im Allgemeinen läßt sich der Grad einer Ringerweiterung nicht definieren, da der Begriff der Dimension für Ringe für gewöhnlich nicht existiert. Speziell für $\mathbb{Z}(\alpha)$ definieren wir ihn aber als Grad des Minimalpolynoms von α.

Kapitel 3

Modulare Berechnung des *ggT*

Wie wir bereits in der Einleitung sehen konnten, lässt sich das Problem des Koeffizientenwachstums mit den nichtmodularen Methoden nicht in den Griff bekommen. Selbst wenn man die überflüssigen Skalare in jedem Schritt des euklidischen Algorithmus aus den Polynomen teilt, wachsen die Koeffizienten exponentiell an.

Die modularen Algorithmen projizieren zuerst die Koeffizienten der gegebenen Polynome in mehrere endliche Körper $\mathbb{Z}_{p_i}$ bzw. endliche Ringe $\mathbb{Z}_{p_i}(\alpha)$. Da hier die Koeffizienten durch die Primzahlen p_i beschränkt sind, läßt sich das Bild des *ggT* ohne Koeffizientenwachstum berechnen. Der eigentliche *ggT* wird dann mit Hilfe des chinesischen Restsatzes rekonstruiert.

Einen wichtigen Schritt hin zu einer modularen *ggT* Berechnung für Polynome aus $\mathbb{Z}(\alpha)[x]$ machte Brown in [6]. Hier entwickelte er einen modularen *ggT* Algorithmus für Polynome aus $\mathbb{Z}[x_1,\ldots,x_n]$. Dieser diente als Grundlage für Langemyr und McCallum, die 1989 in [22] einen modularen Algorithmus zur *ggT* Berechnung für univariate Polynome aus $\mathbb{Z}(\alpha)[x]$ formulierten. Eine Weiterentwicklung dieses Algorithmus lieferte Encarnacion 1995 in [11], indem er das Prinzip der *Rational Reconstruction* (siehe Abschnitt 3.2.4) anwendete.

In Abschnitt 3.1 gehen wir ausführlich auf den Ansatz von Brown ein, wobei wir uns auf univariate Polynome beschränken. Abschnitt 3.2 beschäftigt sich zunächst mit den Besonderheiten der *ggT* Berechnung bei Polynomen über algebraischen

Erweiterungen und stellt danach die Lösungsansätze von Langemyr-McCallum und Encarnacion vor. Ausgehend von der Komplexitätsanalyse dieser beiden Algorithmen, stellen wir abschließend den von uns entwickelten Hybridansatz vor. Er stellt ein Verbindung der beiden ersten Algorithmen dar.

3.1 Polynome über $\mathbb{Z}$

3.1.1 Problemstellung und Bezeichnungen

Seien F_1' und F_2' Polynome aus $\mathbb{Z}[x]$. Gesucht wird ein modularer Algorithmus um den *ggT* $G' \in \mathbb{Z}[x]$ der beiden Polynome zu berechnen. Brown nutzte 1971 [6] folgende Eigenschaft aus, um einen modularen *ggT* für Polynome über $\mathbb{Z}$ zu formulieren. Seien f_1' und f_2' die Leitkoeffizienten der beiden Polynome aus $Z[x]$ und p eine Primzahl, die weder f_1' noch f_2' teilt. Dann gilt, wenn F_1' und F_2' teilerfremd über $\mathbb{Z}/p\mathbb{Z}$ sind, dann sind sie auch teilerfremd über $\mathbb{Z}$. Selbst wenn F_1' und F_2' nicht teilerfremd sind, liefert die Berechnung ihres *ggT* über $\mathbb{Z}/p\mathbb{Z}$ nützliche Informationen bezüglich des *ggT* über $\mathbb{Z}$.

Für jede Primzahl $p \in \mathbb{Z}$ ist der Restklassenring $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$ ein Körper mit p Elementen (vgl. (2.13)). Nach Definition 2.23 existiert ein Homomorphismus $\phi_p : \mathbb{Z} \to \mathbb{F}_p$ mit $\phi_p(x) := x \mod p$. Den von ϕ_p induzierten Homomorphismus von $\mathbb{Z}[x]$ in den faktoriellen Ring $\mathbb{F}_p[x]$ bezeichnen wir ebenfalls mit ϕ_p.

Definition 3.1 *Für $F \in \mathbb{Z}[x]$ führen wir folgende Bezeichnungen ein:*

- *Grad(F): Der Grad von F. Im Pseudocode deg(F).*
- *lc(F): Der Leitkoeffizient (leading coefficient) von F.*
- *monic(F): Das normierte Polynom zu F, $monic(F) = F/lc(F) \in \mathbb{Q}[x]$.*
- *cont(F): Der Inhalt (content) von F ist der* ggT *aller Koeffizienten.*
- *pp(F): Der primitive Teil (primitive part) von F, es gilt $F = cont(F) \cdot pp(F)$.*
- *F heißt primitiv, falls cont(F) = 1.*

Da $\mathbb{Z}$ ein faktorieller Ring ist, folgt mit dem Lemma von Gauß 2.9, dass auch $\mathbb{Z}[x]$ ein faktorieller Ring ist. Somit existiert nach Definition 2.8 für jedes Polynom aus $\mathbb{Z}[x]$ eine eindeutige Zerlegung in Primelemente und der *ggT* ist somit eindeutig bestimmt. Die Einheiten von $\mathbb{Z}[x]$ sind die Einheiten von $\mathbb{Z}$.

3.1.2 Der Algorithmus von Brown

Wir präsentieren hier eine verbesserte Variante des Algorithmus, den Brown in seiner Arbeit [6] vorstellt. Benutzt wird zusätzlich eine Überlegung von Langemyr und McCallum [22], damit der Algorithmus schneller terminiert. Dazu aber später mehr.
Die Berechnung von G' kann grob in vier Phasen unterteilt werden:

1. Initialisierungsphase
2. Modulare Phase
3. Rekonstruktionsphase
4. Verifikationsphase

Gehen wir nun auf die vier Phasen genauer ein:

1. Initialisierungsphase:

In dieser Phase werden die Polynome soweit wie möglich vereinfacht und wichtige Informationen für später gespeichert.
Wie wir aus Definition 3.1 wissen, können wir jedes Polynom darstellen als $F = cont(F) \cdot pp(F)$. Nach Konstruktion sind $cont(F) \in \mathbb{Z}$ und $pp(F) \in \mathbb{Z}[x]$ teilerfremd, also gilt mit Eigenschaft (2.9) folgende nützliche Regel für $F_1', F_2' \in \mathbb{Z}[x]$:

$$ggT(F_1', F_2') = ggT(cont(F_1'), cont(F_2')) \cdot ggT(pp(F_1'), pp(F_2')).$$

Wir berechnen also den Inhalt der beiden Polynome $c_1 = cont(F_1')$, $c_2 = cont(F_2')$ und $c = ggT(c_1, c_2)$ sowie die primitiven Polynome $F_1 = F_1'/c_1$ und $F_2 = F_2'/c_2$. Später werden wir noch die Leitkoeffizienten $f_1 = lc(F_1)$, $f_2 = lc(F_2)$ und $\overline{g} = ggT(f_1, f_2)$ benötigen.

2. Modulare Phase:

Wir formulieren zunächst eine Aussage über die Eignung bestimmter Primzahlen p für unsere Zwecke. Dazu definieren wir

$$G := ggT(F_1, F_2), \quad \tilde{F_1} := \phi_p(F_1)\,, \tilde{F_2} := \phi_p(F_2)\,, G_p := \phi_p(G).$$

Lemma 3.2 *Sei p eine Primzahl, die den Leitkoeffizienten von G nicht teilt. Dann gilt*

$$Grad(ggT(\tilde{F_1}, \tilde{F_2})) \geq Grad(G).$$

Beweis Sei $D \in \mathbb{Z}[x]$ ein gemeinsamer Teiler von F_1 und F_2. Das heißt es gibt $Q_1, Q_2 \in \mathbb{Z}[x]$ mit $F_1 = Q_1 D$ und $F_2 = Q_2 D$. Natürlich sind die Gleichungen auch dann noch gültig, wenn man die Polynome homomorph in den faktoriellen Ring $\mathbb{F}_p[x]$ transformiert. Die transformierten Elemente kennzeichnen wir mit einer Tilde. Unsere Gleichungen lauten nun also: $\tilde{F_1} = \tilde{Q_1}\tilde{D}$ *bzw.* $\tilde{F_2} = \tilde{Q_2}\tilde{D}$. Offensichtlich ist $\tilde{D}$ weiterhin ein gemeinsamer Teiler von $\tilde{F_1}$ und $\tilde{F_2}$, weshalb auch insbesondere G_p ein gemeinsamer Teiler von $\tilde{F_1}$ und $\tilde{F_2}$ sein muss . Damit ist also G_p auf jeden Fall auch ein Teiler von $ggT(\tilde{F_1}, \tilde{F_2})$. Es gilt also

$$Grad(ggT(\tilde{F_1}, \tilde{F_2})) \geq Grad(G_p).$$

Da p den Leitkoeffizienten von G nicht teilt, gilt außerdem

$$Grad(G_p) = Grad(G).$$

Damit folgt die Behauptung. □

Demzufolge lassen wir nur Primzahlen zu, die den *ggT* von f_1 und f_2 nicht teilen, denn diese teilen sicher auch nicht den Leitkoeffizienten von G was in Lemma 3.2 gefordert wird.

Definition 3.3 *Wir nennen eine Primzahl p glücklich, wenn gilt*

$$p \nmid ggT(f_1, f_2), \tag{3.1}$$

$$Grad(ggT(\tilde{F_1}, \tilde{F_2})) = Grad(G). \tag{3.2}$$

Alle anderen Primzahlen nennen wir unglücklich.

Dass der *ggT* der modularen Polynome $\tilde{F_1}$ und $\tilde{F_2}$ echt größeren Grad haben kann als das Bild des *ggT* $G_p = \phi_p(G)$ und warum die Primzahl den *ggT* der Leitkoeffizienten nicht teilen darf, sieht man an folgendem Beispiel:

Beispiel 3.4 *Sei* $F_1 = (3x+4)^2(x+9)(x+11)$ *und* $F_2 = (3x+4)(x+18)^2$.
Folglich gilt $\overline{g} = ggT(f_1, f_2) = ggT(9,3) = 3$ *und* $G = (3x+4)$ *mit* $Grad(G) = 1$.

Für $p = 3$ *ergibt sich*
$\tilde{F_1} = \phi_3(F_1) = x(x+2)$, $\tilde{F_2} = \phi_3(F_2) = x^2$ *und* $\tilde{G} := ggT(\tilde{F_1}, \tilde{F_2}) = x$.
Es gilt $1 = Grad(\tilde{G}) = Grad(G) > Grad(G_3) = 0$.
Wegen $p \mid \overline{g}$ *ist* $p = 3$ *eine unglückliche Primzahl. Bedingung (3.2) ist zwar erfüllt, trotzdem ist der berechnete modulare* ggT $\tilde{G}$ *offensichtlich falsch.*

Für $p = 5$ *ergibt sich*
$\tilde{F_1} = (3x+4)^2(x+4)(x+1)$, $\tilde{F_2} = (3x+4)(x+3)^2$
und $\tilde{G} := ggT(\tilde{F_1}, \tilde{F_2}) = (3x+4)$.
Wegen $1 = Grad(\tilde{G}) = Grad(G)$, *ist* $p = 5$ *eine glückliche Primzahl.*

Für $p = 7$ *ergibt sich*
$\tilde{F_1} = (3x+4)^2(x+2)(x+4)$, $\tilde{F_2} = (3x+4)(x+4)^2$
und $\tilde{G} := ggT(\tilde{F_1}, \tilde{F_2}) = (3x+4)(x+4)$.
Wegen $2 = Grad(\tilde{G}) > Grad(G) = 1$, *ist* $p = 7$ *eine unglückliche Primzahl.*

In unserem Algorithmus möchten wir nur glückliche Primzahlen wählen. Bedingung (3.1) ist leicht zu prüfen. Demnach lassen sich diese unglücklichen Primzahlen sofort ausschließen. Etwas schwieriger ist es mit Bedingung (3.2), da G noch nicht bekannt ist. Nach Lemma 3.2 wissen wir aber, dass der Grad von G minimal ist und in $\mathbb{F}_p[x]$ höchstens ein zu großer *ggT* errechnet wird. Wir gehen also folgendermaßen vor:

Wir berechnen $\tilde{G_p} = ggT(\tilde{F_1}, \tilde{F_2})$ in $\mathbb{F}_p[x]$ mit Hilfe des euklidischen Algorithmus bis $Grad(\tilde{G_p}) \leq min(Grad(F_1), Grad(F_2))$ gilt. Andernfalls verwerfen wir das Ergebnis, da p unglücklich gewesen sein muss. Dann wählen wir die nächste Primzahl q und berechnen ein neues modulares Bild $\tilde{G_q}$ von G. Diese beiden Er-

gebnisse übergeben wir dann an die Rekonstruktionsphase. Erhalten wir im Laufe der Berechnung allerdings einen höhergradigen *ggT* $\tilde{G}_q$ als die vorher berechneten, müssen wir davon ausgehen, dass die verwendete Primzahl q unglücklich war. Dieses Ergebnis können wir dann einfach verwerfen und es mit einer anderen Primzahl weiterversuchen.
Erhalten wir jedoch ein Ergebnis von kleinerem Grad als die bis dahin berechneten *ggT*, so waren alle bisherigen Primzahlen unglücklich und wir müssen unsere Berechnung komplett neu beginnen. Nur das allerletzte Ergebnis kann behalten werden.
Das beschriebene Vorgehen schließt also aus, dass ein modularer *ggT* von zu hohem Grad (d. h. mit nicht-konstantem Zusatzfaktor) in die Berechnung mit einfließt. In [6] konnte Brown beweisen, dass p mit einer Wahrscheinlichkeit von $1/p$ eine unglückliche Primzahl ist. Da wir mit der größten in double Arithmetik darstellbaren Primzahl starten, ist diese Wahrscheinlichkeit äußerst klein.

Weiterhin ist zu beachten, dass der vom euklidischen Algorithmus berechnete *ggT* $\tilde{G_p} \in \mathbb{F}_p[x]$ einheitsnormal ist. Wir erhalten also nach Definition 2.7 ein normiertes Polynom, welches im Allgemeinen Koeffizienten aus $\mathbb{Q}$ besitzt. Da der chinesische Restsatz allerdings nur Zahlen aus $\mathbb{Z}$ rekonstruieren kann, müssen wir $\tilde{G_p}$ mit dem modularen Bild des *ggT* der Leitkoeffizienten von F_1 und F_2 ($\tilde{g} = \phi_p(\overline{g})$) erweitern. Nach einer Variante des Lemmas von Gauß aus [25] handelt es sich dabei um ein Vielfaches des korrekten Leitkoeffizienten. Somit ist gewährleistet, dass die Koeffizienten von $\tilde{G_p}$ aus $\mathbb{Z}$ sind. Wir nennen $\overline{g}$ eine *multiplikative obere Schranke* von $lc(G)$ oder des Nenners von $monic(G)$.
Würde p den *ggT* der beiden Leitkoeffizienten f_1 und f_2 teilen, dann folgte $\tilde{g} = 0$ und damit $\tilde{G} = 0 \cdot ggT(\tilde{F_1}, \tilde{F_2}) = 0$, was natürlich kein korrektes Ergebnis sein kann.

Fassen wir die modulare Phase also noch einmal kurz zusammen:

- Wähle eine ungerade Primzahl p mit $p \nmid \overline{g} = ggT(f_1, f_2)$.
- Berechne $\tilde{g} = \phi_p(\overline{g})$, $\tilde{F_1} = \phi_p(F_1)$ und $\tilde{F_2} = \phi_p(F_2)$.
- Berechne mit Hilfe des euklidischen Algorithmus
  $\tilde{G} = \tilde{g} \cdot ggT(\tilde{F_1}, \tilde{F_2}) \in \mathbb{F}_p[x]$.

- Prüfe anhand des Grades, ob es sich um eine unglückliche Primzahl handelt und verwerfe gegebenenfalls die falschen Ergebnisse.

3. Rekonstruktionsphase:

Angenommen wir haben zwei in Bezug auf F_1 und F_2 glückliche Primzahlen p,q gefunden und konnten homomorphe Bilder $\tilde{G_p}, \tilde{G_q}$ des gesuchten ggT modulo dieser Primzahlen bestimmen. In unserem angegebenen Algorithmus wird p immer eine neue Primzahl sein, während q das Produkt aller bisher verwendeten Primzahlen speichert. Die Bedingung p und q teilerfremd ist dadurch weiterhin erfüllt. Wir können nun ein homomorphes Bild des ggT modulo pq berechnen, indem wir für die Koeffizienten von $\tilde{G_p}$ und $\tilde{G_q}$ den chinesischen Restsatz 2.28 anwenden. Das Vorgehen wird im Beweis zu Satz 2.28 genauer beschrieben.

Führen wir dieses Verfahren für hinreichend viele Primzahlen p_i durch, erhalten wir irgendwann ein modulares Bild des ggT bezüglich eines Moduls $q = \prod_i p_i$, der größer ist als das Doppelte des Maximums der Beträge aller im ggT auftretenden Koeffizienten. Für $G = ggT(F_1, F_2) = \sum_{i=0}^{n} g_i x^i$ gilt also

$$q > 2Max(\mid g_i \mid).$$

Die Koeffizienten von G ändern sich modulo einer so großen Zahl q also gar nicht mehr. Das berechnete modulare Bild $\tilde{G}$ ist demnach identisch mit dem gesuchten ggT G. Genau das wollten wir erreichen.

Doch woher sollen wir wissen, dass q schon groß genug ist?
Brown benutzt in seiner Arbeit eine sehr naive untere Schranke für q. Er rechnet so lange, bis q größer ist als das Doppelte des maximalen Koeffizienten von F_1 und F_2. Es muss also gelten

$$q \geq 2Max(\mid \varphi \mid),$$

wobei sich φ über alle Koeffizienten von F_1 und F_2 erstreckt. Hat q die genannte Schranke erreicht testet Brown ob der ermittelte ggT $\tilde{G}$ die beiden Ausgangspolynome F_1 und F_2 teilt. Wenn nicht, war q noch nicht groß genug, um die Koeffizienten von $\tilde{G}$ darzustellen und es wird eine weitere Primzahl gewählt. Diese

Schranke ist natürlich sehr pessimistisch gewählt, da die Koeffizienten des *ggT* in der Regel kleiner sind als die der Ausgangspolynome. Demnach stellt sich das Ergebnis gewöhnlich viel früher ein, als dies nach der Schranke zu erwarten wäre.

Verbesserung nach Langemyr und McCallum

Wir wenden eine Überlegung von Langemyr und McCallum aus [22] an, um den Algorithmus noch *output sensitiver* zu machen. Das bedeutet, wir rechnen nicht blind weiter bis q die oben genannte Schranke erreicht hat, sondern wir beziehen die Zwischenergebnisse, d.h. die modularen Bilder des *ggT*, in unsere Überlegungen mit ein. Dazu speichern wir vor der Anwendung des chinesischen Restsatzes das bisherige Ergebnis ab und vergleichen es mit dem neuen Ergebnis. Stimmen sie überein, war mit sehr hoher Wahrscheinlichkeit q groß genug um alle Koeffizienten des *ggT* durch die modulare Berechnung unverändert zu lassen. Dieses Ergebnis $G^\star$ übergeben wir als Kandidat für den richtigen *ggT* an Phase 4.

4. Verifikationsphase:

Um sicherzustellen, dass $G^\star$ der gesuchte *ggT* ist prüfen wir noch

$$G^\star \mid F_1 \textit{ und } G^\star \mid F_2. \tag{3.3}$$

Dieser Test und ist nötig, um zwei sehr unwahrscheinliche Fälle auszuschließen:

1. Alle verwendeten Primzahlen waren unglücklich.

2. Es gilt $\phi_q(G) = \phi_{pq}(G)$, obwohl pq noch nicht groß genug ist.

Erfüllt $G^\star$ die Bedingung (3.3), multiplizieren wir $G^\star$ mit dem ggT der Inhalte $c = ggT(cont(F_1), cont(F_2))$, den wir bereits in Phase 1 berechnet haben und erhalten so den endgültigen *ggT*

$$G' = cG^\star.$$

Algorithm 1: **(Brown's algorithm)** Given the polynomials F_1', $F_2' \in \mathbb{Z}[x]$ with $deg(F_1), deg(F_2) \geq 1$. Compute $G' \in \mathbb{Z}[x]$ the greatest common divisor of F_1' and F_2'.

(1) Set $c_1 = cont(F_1')$, $c_2 = cont(F_2'), c = gcd(c_1, c_2)$.
(2) Set $F_1 = F_1'/c_1$, $F_2 = F_2'/c_2$.
(3) Set $f_1 = lc(F_1)$, $f_2 = lc(F_2)$, $\overline{g} = gcd(f_1, f_2)$.
(4) Set n = 0, e = min(deg(F_1),deg(F_2)).
(5) Let p be a new odd prime not dividing $\overline{g}$
(6) Set $\tilde{g} = \phi_p(\overline{g})$, $\tilde{F_1} = \phi_p(F_1)$, $\tilde{F_2} = \phi_p(F_2)$.
(7) Invoke the euclidean algorithm to compute
$\tilde{G} = \tilde{g} \cdot gcd(\tilde{F_1}, \tilde{F_2})$, over $\mathbb{Z}_p[x]$.
(8) If deg($\tilde{G}$) = 0, set G = 1, and skip to step (15).
If deg($\tilde{G}$) > e, p is an unlucky prime, so go back to step (5).
If deg($\tilde{G}$) < e, the former primes were unlucky.
Set $n = 0$, $e = deg(\tilde{G})$.
(9) Set n = n+1.
(10) If $n = 1$, set the tuple $(q, G^\star) = (p, \tilde{G})$ and go back to step (5).
(11) Use Chinese remainder to update the tuple $(q, G^\star)$:
$(q, G^\star) := chinese\_remainder((q, G^\star), (p, \tilde{G}))$.
(12) If the coefficients of $G^\star$ have changed go back to step (5).
(13) If $G^\star \nmid \overline{g} \cdot F_1$ or $G^\star \nmid \overline{g} \cdot F_2$ go back to step (5).
(14) Set G = pp($G^\star$).
(15) Set G' = cG, and return.

3.1.3 Komplexitätsanalyse

Um eine Abschätzung zu erhalten, die die beobachteten Laufzeiten des Algorithmus von Brown gut modelliert, ist es unpassend eine *worst case* Komplexitätsanalyse zu benutzen. Dies würde nämlich bedeuten, dass der Algorithmus zuerst alle unglücklichen Primzahlen benutzt, wohingegen im Allgemeinen keine der gewählten Primzahlen unglücklich ist. Wir gehen in unserer Analyse also davon aus, dass keine unglückliche Primzahl auftritt. Da der Algorithmus, wie bereits erwähnt, *output sensitive* ist, beziehen wir die Größe des Endergebnisses $G = ggT(F_1, F_2)$ in unsere Analyse mit ein.

Um den Aufwand des Algorithmus darzustellen benutzen wir folgende Notationen:

$m = max(Grad(F_1), Grad(F_2))$
$\mathcal{F}$ = Das Maximum der Beträge der Koeffizienten von F_1 und F_2
$\mathcal{D} = \gcd(f_1, f_2)$
$\mathcal{G}$ = Das Maximum der Beträge der Nenner und Zähler von $monic(G) \in \mathbb{Q}[X]$
$g = Grad(G)$

Mit diesen Bezeichnungen ergibt sich als Komplexität für den Algorithmus von Brown:

$$O\left(\log(\mathcal{F}+\mathcal{G})(m-g)g + \log(\mathcal{D}+\mathcal{G})(m\log(\mathcal{F}) + m^2 + g\log(\mathcal{D}+\mathcal{G}))\right) \quad (3.4)$$

Der erste Summand beschreibt die Kosten der Testdivision während der Verifikationsphase. Diese hängen ab vom Grad der Ausgangspolynome und des *ggT*, sowie der jeweiligen Koeffizientengröße. Nach Encarnacion [11] folgt dann weiter:
Der Algorithmus benötigt insgesamt $O(\log(\mathcal{D}+\mathcal{G}))$ Primzahlen. Für jede Primzahl werden die modularen Bilder der Koeffizienten bestimmt $O(m\log(\mathcal{F}))$, der modulare *ggT* berechnet $O\left(m^2\right)$ und der chinesische Restsatz angewendet $O(g\log(\mathcal{D}+\mathcal{G}))$.

3.2 Polynome über algebraischen Erweiterungen

Seien F_1 und F_2 Polynome über einer algebraischen Erweiterung von $\mathbb{Z}$. Die modulare ggT-Berechnung für solche Polynome gestaltet sich etwas schwieriger als der eben präsentierte Algorithmus. Die Grundstruktur von Brown kann zwar übernommen werden, aufgrund des anderen Koeffiziententyps müssen wir jedoch einige Ergänzungen und Abwandlungen vornehmen. Der Grund dafür ist, dass eine algebraische Erweiterung von $\mathbb{Z}$ nur einen Integritätsbereich darstellt, in dem nach Definition 2.1 keine eindeutige Zerlegung in irreduzible Elemente existiert und somit auch kein eindeutiger ggT definiert werden kann. Wir führen zuerst ein paar nützliche Notationen ein, die wir im Folgenden benutzen werden. Die mathematischen Begriffe wurden in Abschnitt 2.4 (Algebraische Erweiterungen) definiert.

3.2.1 Begriffe und Notationen

Mit kleinen griechischen Buchstaben bezeichnen wir ganze algebraische Zahlen und mit O meinen wir den Ring der ganzen algebraischen Zahlen.

Für jedes $\alpha \in O$ existiert ein eindeutiges normiertes Polynom $M \in \mathbb{Z}[t]$, welches α als Nullstelle besitzt. Wir nennen M das *Minimalpolynom* von α. Den Grad von M, der gleichzeitig den Grad der algebraischen Erweiterung angibt, bezeichnen wir mit $n = Grad(M)$ und den Leitkoeffizienten mit $l = lc(M)$.

Für ein gegebenes Minimalpolynom M sprechen wir vom Körper $K = \mathbb{Q}(\alpha)$, der sich aus einer Erweiterung der rationalen Zahlen um eine beliebige Nullstelle von M ergibt. O_K symbolisiert den Unterring von O, der aus den ganzen algebraischen Zahlen von K besteht.

Für α schreiben wir nach Definition 2.36 den Integritätsring der ganzen algebraischen Erweiterung von $\mathbb{Z}$ vom Grad n als

$$\mathbb{Z}(\alpha) = \{a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1} : a_i \in \mathbb{Z}\} \tag{3.5}$$

und definieren

$$(1/a)\mathbb{Z}(\alpha) := \{(1/a)\delta : \delta \in \mathbb{Z}(\alpha)\}. \tag{3.6}$$

Die Koeffizienten unserer Polynome F_1 und F_2 sind also aus $\mathbb{Z}(\alpha)$. Ein Element $a_i \in \mathbb{Z}$ aus (3.5) bezeichnen wir als *ganzzahligen Koeffizienten* oder *skalaren Koeffizienten* der algebraischen Erweiterung von $\mathbb{Z}$.

Wie sehen aber die Koeffizienten der modularen Bilder dieser Polynome aus? Für eine gegebene Primzahl $p \in \mathbb{Z}$ definieren wir $M_p = M \mod p$ als modulares Bild des Minimalpolynoms zu α. Da wir die Aussage von Lemma 2.34 auch auf Ringe übertragen können, gilt $\mathbb{Z}(\alpha) \simeq \mathbb{Z}[t]/\langle M\rangle$ und damit auch $\mathbb{F}_p(\alpha) = \mathbb{Z}/p\mathbb{Z}(\alpha) \simeq \mathbb{F}_p[t]/\langle M_p\rangle$. Wir definieren $\mathbf{R}_p = \mathbb{F}_p[t]/\langle M_p\rangle$ als Koeffiziententyp der modularen Polynome. Sei weiter $\phi_p : \mathbb{Z}(\alpha) \to \mathbf{R}_p$ der Ringhomomorphismus mit $\phi_p : B(\alpha) \mapsto (B(t) \mod p) \mod M_p$. Den von ϕ_p induzierten Homomorphismus von $\mathbb{Z}(\alpha)[x]$ nach $\mathbf{R}_p[x]$ bezeichnen wir ebenfalls mit ϕ_p.

Nach Definition 2.6 schreiben wir $res(A,B)$ für die Resultante von $A,B \in \mathbb{Z}[t]$. Für ein Element $\beta \in \mathbb{Z}(\alpha)$ definieren wir $res(M,\beta) = res(M,B)$, wobei $\beta = B(\alpha)$ mit $B \in \mathbb{Z}[t]$ und $Grad(B) < n$. Gilt $\beta \in \mathbb{Z}$, also $Grad(B) = 0$, setzen wir $res(M,\beta) = \beta$.

Die Diskriminante von M ist definiert als die um $l = lc(M)$ reduzierte Resultante von M mit seiner Ableitung M':

$$D = disc(M) = (-1)^{n(n-1)/2} l^{-1} res(M,M'). \tag{3.7}$$

Weiter bezeichnen wir mit

$$d \text{ die größte ganze Zahl, deren Quadrat } D \text{ teilt.} \tag{3.8}$$

Offensichtlich können wir für ein beliebiges Element $\beta \in K = \mathbb{Q}(\alpha)$ unendlich viele $b \in \mathbb{Z}$ finden, so dass $\beta \in (1/b)\mathbb{Z}(\alpha)$. Das kleinste b das diese Bedingung erfüllt nennen wir den *Nenner* von β. Für ein Polynom $F \in (1/r)\mathbb{Z}(\alpha)[x]$ sagen wir, r ist eine *multiplikative Schranke* aller Nenner von F.

3.2.2 Unterschiede zu Polynomen über $\mathbb{Z}$

Was ist aufgrund des geänderten Koeffizientyps zu beachten?

- Um den *ggT* der Leitkoeffizienten von F_1, F_2 berechnen zu können, müssen wir gewährleisten, dass diese aus $\mathbb{Z}$ stammen. Dazu erweitern wir die Polynome mit dem sogenannten *normalization_factor* (siehe Abschnitt 4.3.1).

- Der Inhalt der Polynome ist nicht definiert. Um jedoch trotzdem überflüssige skalare Faktoren aus den Polynomen entfernen zu können, benutzen wir den Funktor *Scalar_factor*. Die Funktionsweise wird in Abschnitt 4.3.1 im Rahmen der Beschreibung der Klasse `Scalar_factor_traits` erläutert.

- Der Polynomring $\mathbb{Z}(\alpha)[x]$ ist, nach Bemerkung 2.5, ebenfalls nur noch ein Integritätsbereich, d.h. wir können den *ggT* nur bis auf einen konstanten Faktor genau bestimmen.

- Die multiplikative obere Schranke des Nenners von G muss um den *Denominator for Algebraic Integer* erweitert werden (siehe Abschnitt 3.2.3).

- Die *ggT*-Berechnung in $\mathbf{R}_p[x]$ funktioniert nicht immer, da $\mathbf{R}_p$ allgemein kein Körper ist. Um einen normierten *ggT* zu erhalten, kann es vorkommen, dass wir einen Nullteiler invertieren müssen. Auf dieses Problem werden wir in Abschnitt 4.3.2 genauer eingehen.

3.2.3 Der Algorithmus von Langemyr und McCallum

Langemyr und McCallum stellten 1989 in ihrem paper *„The Computation of Polynomial Greatest Common Divisors over an Algebraic Number Field“* [22] einen Algorithmus vor, um den *ggT* von zwei Polynomen über einer algebraischen Erweiterung von $\mathbb{Z}$ von beliebigem Grad zu bestimmen. Um das Terminieren des Algorithmus gemäß des chinesischen Restsatzes sicher zu stellen, benutzen sie den *Denominator for Algebraic Integer* auf den wir im Folgenden ausführlicher eingehen.

Denominator for Algebraic Integer

Da der chinesische Restsatz nur Koeffizienten aus $\mathbb{Z}$ rekonstruieren kann, mussten wir in Schritt (7) von Algorithmus 1 den berechneten normierten $ggT\ \tilde{G}$ mit einer multiplikativen oberen Schranke der Nenner erweitern. Für Polynome aus $\mathbb{Z}[x]$ ist das einfach der *ggT* der beiden Leitkoeffizienten.

Diese Schranke ist für Polynome über algebraischen Erweiterungen von $\mathbb{Z}$ nicht ganz so einfach zu bestimmen. Wie das folgende Beispiel zeigt, reicht es in einigen Fällen nicht aus mit dem *ggT* der Leitkoeffizienten zu erweitern.

Beispiel 3.5 *Sei* $G = (1+\sqrt{5})x+2 \in \mathbb{Z}(\sqrt{5})[x]$ *der* ggT *von*

$$F_1 = ((1+\sqrt{5})x+2)\cdot((1-\sqrt{5})x+4) = 4x^2-2(3+\sqrt{5})x-8 \quad \in \mathbb{Z}(\sqrt{5})[x],$$
$$F_2 = ((1+\sqrt{5})x+2)\cdot((1-\sqrt{5})x+6) = 4x^2-4(2+\sqrt{5})x-12 \quad \in \mathbb{Z}(\sqrt{5})[x].$$

Unser Algorithmus eliminiert zuerst die überflüssigen Skalare

$$F_1 = F_1/scalar\_factor(F_1) = F_1/2 = 2x^2-(3+\sqrt{5})x-4 \quad \in \mathbb{Z}(\sqrt{5})[x],$$
$$F_2 = F_2/scalar\_factor(F_2) = F_2/4 = x^2-(2+\sqrt{5})x-3 \quad \in \mathbb{Z}(\sqrt{5})[x].$$

Danach wird der ggT *der beiden Leitkoeffizienten bestimmt*

$$\overline{g} = ggT(lc(F_1), lc(F_2)) = ggT(2,1) = 1.$$

Folglich wird der berechnete ggT *in der modularen Phase mit* $\phi_p(1) = 1$ *erweitert und das Endergebnis ist somit ein normierter* ggT. *Das normierte assoziierte Polynom zu G sieht aber folgendermaßen aus:*

$$G' = G\cdot(-(1-\sqrt{5})/4) = x-1/2+\sqrt{5}/2 \notin \mathbb{Z}(\sqrt{5})[x].$$

Um sicher zu stellen, dass der chinesische Restsatz den *ggT* auch rekonstruieren kann, müssen wir die multiplikative Schranke $\overline{g}$ also noch um einen Faktor erweitern . Diesen Faktor nennen wir *Denominator for Algebraic Integer (D)*.

Ergebnisse aus der Literatur

Nun wollen wir eine solche Schranke für den *ggT* von F_1 und F_2 finden, die möglichst klein und leicht zu berechnen ist. Dafür tragen wir ein paar Ergebnisse

zusammen, die ausführlich von Encarnacion in [11], Langemyr und McCallum in [22] und Weinberger und Rothschild in [28] diskutiert werden.

Unser erstes Problem ist, den Nenner eines normierten Polynoms aus $\mathbb{Z}(\alpha)[x]$ zu finden. Die Lösung liefert uns das folgende Lemma aus [11].

Lemma 3.6 *Ist $\tilde{F}$ das normierte assoziierte Polynom von $F \in \mathbb{Z}(\alpha)[x]$, dann gilt*

$$\tilde{F} \in (1/f)\mathbb{Z}(\alpha)[x],$$

wobei $f = res(M, lc(F))$.

Über die anderen normierten Teiler von F haben Weinberger und Rothschild in [28] folgendes herausgefunden:

Theorem 3.7 ***(Weinberger-Rothschild)*** *Sei α eine algebraische Zahl und $\tilde{F} \in (1/f)\mathbb{Z}(\alpha)[x]$ ein normiertes Polynom. Ist G ein normierter Teiler von $\tilde{F}$ über K, dann gilt*

$$G \in (1/fd)\mathbb{Z}(\alpha)[x],$$

mit d aus (3.8).

Insgesamt erhält man folgende Aussage über den normierten *ggT* und dessen Nenner:

Theorem 3.8 *Sei $F_1, F_2 \in \mathbb{Z}(\alpha)[x]$ und $f_1 = res(M, lc(F_1)), f_2 = res(M, lc(F_2))$. Weiter sei G der normierte* ggT *von F_1, F_2 über $\mathbb{Q}(\alpha)[x]$. Dann gilt*

$$G \in (1/b)\mathbb{Z}(\alpha)[x], \text{ mit } b = ggT(f_1, f_2)d.$$

Da aus [24] bekannt ist, dass d sehr schwierig zu berechnen ist werden wir in der Praxis für den *Denominator for Algebraic Integer $D = disc(M)$* benutzen. Es gilt, nach Definition, $d^2 \mid D$ und damit ist D eine multiplikative obere Schranke zu d (alle Teiler von d teilen auch D). Für Polynome über $\mathbb{Z}(\alpha)[x]$ gilt also

$$\overline{g} = ggT(f_1, f_2)D. \tag{3.9}$$

Offensichtlich ist $\overline{g}$ ein schlechte obere Schranke für den eigentlichen Nenner $ggT(f_1, f_2)d$. Auf dieses Problem werden wir im Verlauf dieser Arbeit noch eingehen.

Bedeutung für unsere Anwendung

Da wir in Schritt 1 der Algorithmen die Polynome $F_1, F_2 \in \mathbb{Z}(\alpha)[x]$ so erweitern, dass $lc(F_1), lc(F_2) \in \mathbb{Z}$ gilt, folgt nach Definition für f_1, f_2 :

$$f_1 = res(M, lc(F_1)) = lc(F_1) \text{ und } f_2 = res(M, lc(F_2)) = lc(F_2).$$

Bleibt noch der *Denominator for Algebraic Integer (D)* zu bestimmen. Allgemein gilt nach (3.7) $D = disc(M) = (-1)^{n(n-1)/2} l^{-1} res(M, M')$. Da unser Hauptinteresse ganzen algebraischen Erweiterungen vom Grad 2 gilt, sehen wir uns diesen Spezialfall etwas genauer an:

Es gilt also $n = Grad(M) = 2$. Da es sich um eine ganze algebraische Erweiterung handelt, ist der Leitkoeffizient des Minimalpolynoms nach Definition 2.36 $l = 1$. Insgesamt hat das Minimalpolynom die Gestalt $M = x^2 - r$, mit $\alpha = \sqrt{r}$. Damit folgt für den *Denominator for Algebraic Integer*:

$$\begin{aligned} D &= (-1)^{n(n-1)/2} l^{-1} res(M, M') = -1 \cdot res(M, M') \\ &= -1 \cdot \begin{vmatrix} 1 & 0 & -r \\ 2 & 0 & 0 \\ 0 & 2 & 0 \end{vmatrix} = -1 \cdot (-4r) = 4r \end{aligned}$$

In unserem Beispiel 3.5 rechnen wir mit Polynomen aus $\mathbb{Z}(\sqrt{5})[x]$. Der *Denominator for Algebraic Integer* ist hier also $D = 4 \cdot 5 = 20$. Wir erweitern unsere multiplikative Schranke $\overline{g}$ daher um den Faktor $D = 20$ und erhalten somit

$$G^* = 20 \cdot G' = 20 \cdot (x - 1/2 + \sqrt{5}/2) = 20x - 10 + 10\sqrt{5} \in \mathbb{Z}(\sqrt{5})[x].$$

Mit Hilfe des chinesischen Restsatzes lässt sich dieses Polynom korrekt rekonstruieren. Entfernen wir noch die überflüssigen Skalare, so erhalten wir den gesuchten $ggT\ G = 2x - 1 + \sqrt{5}$ mit normalisiertem Leitkoeffizienten.

Wichtig bei der Wahl der Primzahl ist, dass diese kein Teiler des *Denominator for Algebraic Integer* ist. Denn angenommen, die gewählte Primzahl p teilt D, dann folgt wegen (3.9) $\tilde{g} = \phi_p(\overline{g}) = 0$ und damit, in Schritt (7) von Algorithmus 2, $\tilde{G} = 0$. Das kann natürlich kein korrektes Ergebnis sein.

Algorithm 2: **(Langemyr and McCallum)** Given the polynomials $F_1, F_2 \in \mathbb{Z}(\alpha)[x]$ compute $G^\star \in \mathbb{Z}(\alpha)[x]$, the (up to constant factor) greatest common divisor of F_1 and F_2.

(1) Set $F_i = normalization\_factor(lc(F_i)) \cdot F_i$.
(2) Set $F_i = F_i / scalar\_factor(F_i)$.
(3) Set $f_i = lc(F_i) \in \mathbb{Z}$.
(3a) Set $D = denominator\_for\_algebraic\_integers(\mathbb{Z}(\alpha))$.
(3b) Set $\overline{g} = gcd(f_1, f_2)D$.
(4) Set $n = 0$, $e = min(deg(F_1), deg(F_2))$.
(5) Select p, a new odd prime not dividing $\overline{g}$.
(6) Set $\tilde{g} = \phi_p(\overline{g}) \in \mathbf{R}_p$, $\tilde{F}_i = \phi_p(F_i) \in \mathbf{R}_p[x]$.
(7) Invoke the euclidean algorithm to compute
$\tilde{G} = \tilde{g} \cdot gcd(\tilde{F}_1, \tilde{F}_2) \in \mathbf{R}_p[x]$.
(8) If $deg(\tilde{G}) = 0$, return $G^\star = 1$.
If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5).
If $deg(\tilde{G}) < e$, the former primes were unlucky.
Set $n = 0$, $e = deg(\tilde{G})$.
(9) Set $n = n + 1$.
(10) If $n = 1$, set the tuple $(q, G^\star) = (p, \tilde{G})$ and go back to step (5).
(11) Use Chinese remainder to update the tuple (q, G^*):
$(q, G^\star) := chinese\_remainder((q, G^\star), (p, \tilde{G}))$.
(12) If the coefficients of $G^\star$ have changed go back to step (5).
(13) Set $G^\star = G^\star / scalar\_factor(G^\star)$.
(14) If $G^\star \nmid F_1$ or $G^\star \nmid F_2$ go back to step (5).
(15) return $G^\star$.

3.2.4 Der Algorithmus von Encarnacion

Wie wir in Abschnitt 3.2.3 gezeigt haben, benötigen Langemyr und McCallum die multiplikative Schranke $\overline{g}$ um sicher zu stellen, dass der zu berechnende ggT $G^\star$ aus $\mathbb{Z}(\alpha)[x]$ ist. Nur so kann der chinesische Restsatz die Koeffizienten von $G^\star$ rekonstruieren.

Encarnacion benutzt in seinem paper *„Computing GCDs of Polynomials over Algebraic Number Fields“* [11] ein anderes Verfahren. Anstatt ein zum *ggT* assoziiertes Polynom aus $\mathbb{Z}(\alpha)[x]$ zu berechnen, versucht er den normierten *ggT* aus $\mathbb{Q}(\alpha)[x]$ zu rekonstruieren. Um die rationalen Zahlen in dieser Darstellung zu finden, benutzt er den sogenannten *Rational Reconstruction* Algorithmus von Wang [26]. Schauen wir uns an, was passiert, wenn wir die berechneten *ggT* in $\mathbf{R}_p[x]$ nicht mit $\overline{g}$ multiplizieren.

Der chinesische Restsatz wird nun versuchen das zu $G^\star$ assoziierte Polynom $G = monic(G^\star) \in \mathbb{Q}(\alpha)[x]$ zu rekonstruieren, was, wie bereits erwähnt, nicht möglich ist. Stattdessen erhalten wir ein Polynom $G' \in \mathbb{Z}(\alpha)[x]$, dessen ganzzahlige Koeffizienten in derselben Restklasse sind, wie die entsprechenden Koeffizienten von $G \in \mathbb{Q}(\alpha)[x]$. Der *Rational Reconstruction* Algorithmus ermöglicht es nun, aus den Koeffizienten von G' die rationalen Koeffizienten von G zu rekonstruieren.

Formulieren wir dieses Problem konkret für einen ganzzahligen Koeffizienten $u \in \mathbb{Z}$ von G':
Sei $n/d \in \mathbb{Q}$, $d > 0$ und n,d teilerfremd. Die ganze Zahl m sei das Produkt der gewählten Primzahlen und $u = n/d \mod m$. Dann sind u und n/d Elemente derselben Restklasse. Das Problem der *Rational Reconstruction* ist also: gegeben u und m, finde n/d. Die rationale Zahl n/d ist der zu u entsprechende Koeffizient des normierten *ggT* G.

Der *Rational Reconstruction* Algorithmus von Wang (*RR*)

Wie in der Einleitung beschrieben, seien $n,d \in \mathbb{Z}$, $d > 0$, $m \in \mathbb{N}$ mit $ggT(n,d) = 1$ und $u = n/d \mod m$. Also ist n/d die rationale Zahl, die wir rekonstruieren möchten.
Um den RR Algorithmus von Wang zu verstehen, wiederholen wir noch einmal die Ergebnisse des erweiterten Euklidischen Algorithmus (*EEA*, Satz 2.20).
Für Eingabewerte m und u mit $m > u \geq 0$ errechnet der EEA eine Sequenz von Tripeln $(r_i,s_i,t_i) \in \mathbb{Z}^3$ mit $i = 0,1,\ldots,l,l+1$ wobei $r_{l+1} = 0$ und $s_i m + t_i u = r_i$ gilt. Hieraus folgt $r_i/t_i \equiv u \mod m$, für alle i mit $ggT(m,t_i) = 1$.

Auch hier stellt sich die Frage, wie groß das Produkt der Primzahlen sein muss um die gesuchte rationale Zahl eindeutig bestimmen zu können. In [26] konnte Paul S. Wang folgendes beweisen:

Lemma 3.9 *Für gegebene Zahlen* $u,m \in \mathbb{Z}$ *sind* $n \in \mathbb{Z}$ *und* $d \in \mathbb{N}$ *mit* $n/d \equiv u \mod m$ *eindeutig, wenn*

$$| n | < \sqrt{m/2} \quad \textit{und} \quad | d | < \sqrt{m/2}$$

gilt.

Beweis Angenommen es existiert eine weitere rationale Zahl n'/d' mit $ggT(n',d') = 1$, so dass $| n' | < \sqrt{m/2}$, $| d' | < \sqrt{m/2}$ und $n/d \equiv n'/d' \mod m$ gilt.
Dann folgt

$$d'n \equiv dn' \mod m \tag{3.10}$$

$$\Rightarrow d'n - dn' = i \cdot \text{m für ein i } \in \mathbb{Z}. \tag{3.11}$$

Weiterhin gilt

$$| nd' | < \frac{m}{2} \quad \text{und} \quad | dn' | < \frac{m}{2}$$

und damit

$$-m < (d'n - dn') < m. \tag{3.12}$$

Aus (3.11) und (3.12) folgt i = 0 und damit $n = n'$ und $d = d'$.

□

Demnach ist die Zahl n/d eindeutig, wenn $m > 2max(| n |,d)^2$ gilt.

In [27] beweisen Wang, Guy und Davenport, dass der Algorithmus die Zahl n/d sicher findet, falls sie existiert. Allerdings ist eine solche rationale Zahl nicht immer vorhanden, wie das nächste Beispiel zeigt.

Beispiel 3.10 *Für* $m = 11$ *erhalten wir* $| n |, d \leq 2$ $(n \in \mathbb{Z},\ d \in \mathbb{N})$ *und somit folgende rationale Zahlen:*

$$\begin{array}{lll} 0/1 \equiv 0, & 1/1 \equiv 1, & -1/1 \equiv 10, \\ 2/1 \equiv 2, & -2/1 \equiv 9, & 1/2 \equiv 6, \\ -1/2 \equiv 5 & & \end{array}$$

Eine rationale Darstellung für $3,4,7$ *und* 8 *existiert demnach nicht.*

Wang nutzt diese Beobachtungen für seinen Algorithmus, indem er folgendermaßen vorgeht. Für gegebene Zahlen u und m berechnet der Algorithmus die bekannten Grenzen $N = D = \lfloor \sqrt{m/2} \rfloor$ und führt den erweiterten euklidischen Algorithmus (EEA) mit den Eingabewerten $m > u$ aus bis $r_{i-1} > N \geq r_i$ erfüllt ist. Dann wird geprüft, ob $\mid t_i \mid \leq D$ gilt. Wenn ja, gibt Wang's RR "true" und $n/d = r_i/t_i$ zurück, ansonsten war m noch nicht groß genug es wird "false" zurückgegeben.

Um zu zeigen, dass der Algorithmus von Wang noch nicht vollständig ist, machen Collins und Encarnacion in [8] auf folgendes Gegenbeispiel aufmerksam. Angewendet auf $m = 12$ und $u = 5$ errechnet der Algorithmus $(n,d) = (-2,2)$. Allerdings gilt $-2/2 \not\equiv 5 \mod 12$, da 2 in $\mathbb{Z}/12\mathbb{Z}$ nicht invertierbar ist. Offensichtlich ist auch $-1 \not\equiv 5 \mod 12$. Demnach muss also noch zusätzlich

$$ggT(d,m) = 1$$

getestet werden, was bedeutet, dass d in $\mathbb{Z}/m\mathbb{Z}$ invertierbar ist. Die Invertierbarkeit von d kann aber noch etwas effizienter sichergestellt werden.

Lemma 3.11 *Sei $n,d \in \mathbb{Z}$, $d > 0$ das Ergebnis von Wang's Algorithmus mit $u \equiv n/d \mod m$. Dann gilt:*

$$ggT(d,m) = ggT(n,d)$$

Beweis Sei $c = (n-du)/m$. Angenommen $ggT(d,c) \neq 1$. Dann existiert p prim mit $p \mid d$ *und* $p \mid c$. Daraus folgt $p \mid cm$ also auch $p \mid (n-du)$. Wegen $p \mid d$ gilt auch $p \mid n$. Dies ist aber ein Widerspruch zu ggT(n, d) = 1, da (n,d) das Ergebnis aus Wang's Algorithmus ist. Also gilt $ggT(d,c) = 1$ und damit folgt

$$ggT(d,m) \;=\; ggT(d,cm) \;=\; ggT(d,n \;-\; du) \overset{(2.8)}{=} ggT(d,n). \qquad \square$$

Der Test $ggT\,(n,d) = 1$ ist günstiger als $ggT\,(m,d) = 1$, da nach Voraussetzung $n,\ d\ < \sqrt{m/2}$ gilt.

Insgesamt muss die gesuchte rationale Zahl n/d also folgende Bedingungen erfüllen:

$$n/d \equiv u \ (mod\ m), \tag{3.13}$$

$$0 \leq \mid n \mid < \sqrt{m/2}, \quad 0 < d < \sqrt{m/2}, \tag{3.14}$$

$$ggT(n,d) = 1. \tag{3.15}$$

Den Pseudocode des RR Algorithmus präsentieren wir bereits in diesem Abschnitt, auf die Implementierungsdetails gehen wir unter Abschnitt 4.3 ein.

Algorithm 3: **(Wang's Rational Reconstruction)** Given $m,\ u \in \mathbb{Z}$, $m > u \geq 0$, m odd. $N = D = \lfloor \sqrt{m/2} \rfloor$. Compute $n, d \in \mathbb{Z}$, with $\mid n \mid \leq N,\ 0 < d \leq D$, $ggT(n,d) = 1$ and $n/d \equiv u \ (mod\ m)$. If n/d exists say TRUE otherwise FALSE.

Set $(r_0, t_0) = (m, 0)$;
Set $(r_1, t_1) = (u, 1)$;
while $r_1 > N$ **do**
    Set $q = \lfloor r_0/r_1 \rfloor$;
    Set $(r_0, r_1) = (r_1, r_0 - qr_1)$;
    Set $(t_0, t_1) = (t_1, t_0 - qt_1)$;
end
Set $(n, d) = (r_1, t_1)$;
if $d < 0$ **then**
    Set $(n, d) = (-n, -d)$;
end
if $d \leq D$ *and* $ggT(n,d) = 1$ **then**
    return TRUE and (n, d);
else
    return FALSE;
end

Beispiel 3.12 *Für $m = 19$ rechnet der Algorithmus von Wang mit $N = D = 3$ und rekonstruiert folgende rationale Zahlen. (Ein F steht für Rückgabe FALSE.)*

$u =$	1	2	3	4	5	6	7	8	9
$n/d =$	1	2	3	F	F	$-1/3$	$2/3$	$-3/2$	$-1/2$
$u =$	10	11	12	13	14	15	16	17	18
$n/d =$	$1/2$	$3/2$	$-2/3$	$1/3$	F	F	-3	-2	-1

Schauen wir uns den Verlauf des Algorithmus am Beispiel $u = 13$ an:

$$\begin{aligned}
(r_0, t_0) &= (19, 0) \\
(r_1, t_1) &= (13, 1) \\
q &= \lfloor r_0 / r_1 \rfloor = 1 \\
(r_0, r_1) &= (13, 6) \\
(t_0, t_1) &= (1, -1) \qquad (\mathbf{r_1} > \mathbf{N}) \\
q &= \lfloor r_0 / r_1 \rfloor = 2 \\
(r_0, r_1) &= (6, 1) \\
(t_0, t_1) &= (-1, 3) \qquad (\mathbf{r_1} < \mathbf{N}) \\
(n, d) &= (1, 3)
\end{aligned}$$

Da $d \leq D$ und $ggT(n,d) = 1$ gibt der Algorithmus TRUE und $(n,d) = (1,3)$ zurück.

Komplexität des RR Algorithmus

Die Laufzeit von Wang's Algorithmus ist offensichtlich die gleiche wie die des erweiterten Euklidischen Algorithmus (Satz 2.20). Um diese zu analysieren, seien $a > b > 0$ zwei ganze Zahlen mit höchstens m Bits, deren ggT berechnet werden soll. Uns interessiert im Wesentlichen, wie viele Divisionen mit Rest durchgeführt werden und wieviel eine einzelne Division kostet.

Bei der Laufzeitanalyse stellt sich heraus, dass der schlimmste Eingabefall zwei aufeinander folgende Fibonacci-Zahlen sind. Diese Tatsache konnte Gabriel Lamé in [20] beweisen. Bei aufeinander folgenden Fibonacci-Zahlen ergibt sich als Rest

immer die nächst kleinere Fibonacci-Zahl. Die Anzahl der benötigten Divisionen beträgt demnach im schlimmsten Fall $k = O(m)$.
Der Aufwand für die Division zweier m Bit Zahlen beträgt $O(m(d+1))$, wobei d die Anzahl der Bits des Quotienten angibt (vergleiche von zur Gathen [12], Seite 54).

Betrachten wir nun die Rechnung $ggT(a,b)$. Wir bezeichnen mit $a_0,\ldots,a_k$ die Sequenz der Reste und mit $m_0,\ldots,m_k$ deren Bitlänge. Dann ergibt sich für die Laufzeit des erweiterten euklidischen Algorithmus:

$$\begin{aligned} O\Big(\sum_{i=0}^{k-1} m_i(m_i - m_{i+1} + 2)\Big) &\subseteq O\Big(m \sum_{i=0}^{k-1} (m_i - m_{i+1} + 2)\Big) \\ &\subseteq O\big(m(m_0 - m_k + 2k)\big) \\ &\subseteq O\big(m(m_0 + 2k)\big) \\ &\subseteq O\big(m^2\big). \end{aligned}$$

Lemma 3.13 *Die Laufzeit des Rational Reconstruction Algorithmus von Wang ist $O\big(m^2\big)$. Er verläuft somit quadratisch in der Größe der Eingabewerte bzw. der zu rekonstruierenden Koeffizienten.*

Bedeutung für unsere Anwendung

In Algorithmus 4 berechnet der *Rational Reconstruction* Algorithmus das Polynom $R = G^\star/d \in \mathbb{Q}(\alpha)[x]$, bestehend aus dem Zählerpolynom $G^\star \in \mathbb{Z}(\alpha)[x]$ und dem Hauptnenner $d \in \mathbb{Z}$. Vergleiche hierzu die Beschreibung der Klasse `Wang_traits` unter 4.3.1. Da wir nur mit $G^\star$ weiterrechnen, müssen wir vor der Testdivision (Schritt (13) in Algorithmus 4) die Polynome F_1 und F_2 mit dem Hauptnenner d multiplizieren.

Ausserdem muss für den *Rational Reconstruction* Algorithmus sichergestellt sein, dass das Produkt der Primzahlen keinen Nenner der rationalen Zahlen teilt, die wir rekonstruieren möchten. Aus diesem Grund prüfen wir in Schritt (5) von

Algorithmus 4, dass p weder f_1, f_2 noch D teilt, welches Vielfache der möglichen Nenner sind.

Algorithm 4: **(Encarnacion)** Given the polynomials $F_1, F_2 \in \mathbb{Z}(\alpha)[x]$. Compute $G^\star \in \mathbb{Z}(\alpha)[x]$, the (up to constant factor) greatest common divisor of F_1 and F_2.

---

(1) Set $F_i = normalization\_factor(lc(F_i)) \cdot F_i$.
(2) Set $F_i = F_i / scalar\_factor(F_i)$.
(3) Set $f_i = lc(F_i) \in \mathbb{Z}$.
(3a) Set $D = denominator\_for\_algebraic\_integers(\mathbb{Z}(\alpha))$.
(3b) `dispensed` // computation of $\overline{g}$
(4) Set $n = 0$, $e = min(deg(F_1), deg(F_2))$.
(5) Select p, a new odd prime not dividing f_1, f_2 or D.
(6) Set $\tilde{F}_i = \phi_p(F_i) \in \mathbf{R}_p[x]$.
(7) Invoke the euclidean algorithm to compute
$\tilde{G} = gcd(\tilde{F}_1, \tilde{F}_2) \in \mathbf{R}_p[x]$.
(8) If $deg(\tilde{G}) = 0$, return $G^\star = 1$.
If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5).
If $deg(\tilde{G}) < e$, the former primes were unlucky.
Set $n = 0$, $e = deg(\tilde{G})$.
(9) Set $n = n + 1$.
(10) If $n = 1$, set the tuple $(q, G^\star) = (p, \tilde{G})$ and go back to step (5).
(11) Use Chinese remainder to update the tuple (q, G^*):
$(q, G^\star) := chinese\_remainder((q, G^\star), (p, \tilde{G}))$.
(11a) Try to use the RR algorithm to compute $(G^\star, d) := RR(q, G^\star)$,
where $G^\star \in \mathbb{Z}(\alpha)[x]$ and $d \in \mathbb{Z}$ the found denominator.
(12) If RR fails or $G^\star$ has changed go back to step (5).
(13) If $G^\star \nmid d \cdot F_1$ or $G^\star \nmid d \cdot F_2$ go back to step (5).
(14) return $G^\star$.

3.2.5 Komplexitätsanalyse der beiden Algorithmen

Der Vorteil von Encarnacions Algorithmus besteht offensichtlich darin, dass der gesuchte ggT nicht mit der ungenauen multiplikativen Schranke $\overline{g}$ erweitert wird, wie es im Ansatz von Langemyr und McCallum (Algorithmus 2) geschieht. Das hat nämlich den Nachteil, dass der gesuchte ggT um unnötig viele Bits vergrößert wird, die dann mit Hilfe des chinesischen Restsatzes wiederhergestellt werden müssen. Andererseits leidet der Algorithmus von Encarnacion unter den zusätzlichen Kosten der *Rational Reconstruction*.

Wir präsentieren im Folgenden die Ergebnisse von Encarnacion, der in [11] beide Ansätze diskutiert hat. Wie bei der Komplexitätsanalyse des Algorithmus von Brown ist es unpassend eine *worst case* Analyse zu verwenden. Auch Encarnacion geht deshalb davon aus, dass keine unglückliche Primzahl auftritt. Da die Algorithmen, wie bereits erwähnt, *output sensitive* sind, bezieht Encarnacion die Größe des Endergebnisses in seine Analyse mit ein.

Um den Aufwand der Algorithmen darzustellen benutzen wir die Abkürzungen der Algorithmen und folgende Notationen:

$m = max(Grad(F_1), Grad(F_2))$

$n =$ Grad(M), Grad der algebraischen Erweiterung

$\mathcal{D} = \gcd(f_1, f_2)D$

$\mathcal{M} =$ Maximum der Beträge der Koeffizienten des Minimalpolynoms $M \in \mathbb{Z}[X]$.

$\mathcal{G} =$ Maximum der Beträge der Nenner und Zähler von $monic(G) \in \mathbb{Q}(\alpha)[X]$

Encarnacion zählt in seiner Analyse die Anzahl der Maschinenwortoperationen, welche die Algorithmen benötigen und geht davon aus, dass diese arithmetischen Operationen in konstanter Zeit ausführbar sind.

Komplexität von Langemyr und McCallum (Algorithmus 2)

Der Algorithmus von Langemyr und McCallum berechnet den *ggT* von F_1 und F_2, mit den oben getroffenen Annahmen in

$$O\left(mn\log^2(\mathcal{D}+\mathcal{G})+m^2n^2(n\log^2(\mathcal{M})+\log^2(m\mathcal{G})+\log(\mathcal{D}+\mathcal{G}))\right) \quad (3.16)$$

Maschinenwortoperationen. Dieser Aufwand setzt sich zusammen aus $O(\log(\mathcal{D}+\mathcal{G}))$ modularen *ggT*-Berechnungen, wobei jede einzelne $O(m^2n^2)$ kostet. Der chinesische Restsatz wird ebenfalls $O(\log(\mathcal{D}+\mathcal{G}))$ mal angewendet. Eine Ausführung kostet $O(mn\log(\mathcal{D}+\mathcal{G}))$. Nach Encarnacion kostet die Testdivision zum Schluß $O(m^2n^2(n\log^2(\mathcal{M})+\log^2(m\mathcal{G})))$. Aufgrund der Annahme, dass nur glückliche Primzahlen gewählt werden, wird sie nur einmal ausgeführt.

Komplexität von Encarnacion (Algorithmus 4)

Der Algorithmus von Encarnacion berechnet den *ggT* von F_1 und F_2, wenn keine unglückliche Primzahl gewählt wird in

$$O\left(mn\log^3(\mathcal{G})+m^2n^2(n\log^2(\mathcal{M})+\log^2(m\mathcal{G}))\right) \quad (3.17)$$

Maschinenwortoperationen. Wir berechnen $O(\log(\mathcal{G}))$ modulare *ggT*, die jeweils $O(m^2n^2)$ Operationen kosten. Zusammen mit den Kosten für den chinesischen Restsatz und die Testdivision erhalten wir den zweiten Summanden der Gesamtkomplexität. Der erste Summand stellt den Aufwand des *rational Reconstruction* Algorithmus dar. Dieser wird ebenfalls $O(\log(\mathcal{G}))$ mal auf $O(mn)$ Koeffizienten der Größe $O(\log(\mathcal{G}))$ angewendet. Da der *Rational Reconstruction* Algorithmus nach Lemma 3.13 quadratisch in der zu rekonstruierenden Koeffizientengröße ist, ergibt sich insgesamt eine Komplexität von $O(mn\log^3(\mathcal{G}))$.

Vergleich der beiden Algorithmen

Wie erwartet ist die Komplexität von Encarnacions Algorithmus unabhängig von der multiplikativen Schranke der Nenner $\mathcal{D}$. Aufgrund des *Rational Reconstruction* Algorithmus wirkt sich hier allerdings die Koeffizientengröße des *ggT* $\mathcal{G}$ kubisch aus. Wir können also annehmen, dass Encarnacions Algorithmus dem von Langemyr und McCallum überlegen ist, falls $\mathcal{G}$ im Vergleich zu $\mathcal{D}$ sehr klein ist.

In unserer Anwendung ist dies allerdings nicht der Fall. Wie erwähnt arbeiten wir im Allgemeinen mit Polynomen über einer algebraischen Erweiterung von kleinem Grad ($n = 2$). Dies hat zur Folge, dass $\mathcal{D}$ nicht sehr groß werden kann (vergleiche Abschnitt 3.2.3) und insgesamt die Kosten der beiden Algorithmen von der Koeffizientengröße $\mathcal{G}$ dominiert werden.
Für Erweiterungen vom Grad 2 können wir also erwarten, dass der Algorithmus von Langemyr und McCallum besser arbeitet als der von Encarnacion. Diese Vermutung wird von den Benchmarks in Kapitel 5 gestützt.

3.2.6 Eigener Hybridansatz

Wie die Komplexitätsanalyse 3.2.5 gezeigt hat, haben beide Algorithmen ihre Schwächen. Der Ansatz von Encarnacion leidet unter den Kosten des Algorithmus von Wang, der in jedem Durchlauf ausgeführt wird. Die Methode von Langemyr und McCallum ist zwar effizienter, durch die Multiplikation mit der ungenauen Schranke $ggT(f_1, f_2)D$ wird das Ergebnis allerdings unnötig vergrößert. Um diese überflüssigen Bits zu rekonstruieren, benötigt man natürlich weitere Primzahlen, wodurch die Rechnung teurer wird. Im Laufe unserer Tests und Benchmarks stellte sich heraus, dass der *Denominator for Algebraic Integer* in der Regel überflüssig ist. In allen von uns erzeugten Beispielpolynomen, sowie in den Benchmarks der Dissertation von Michael Hemmer [15], wurde der zusätzliche Faktor D nur einmal benötigt.

Wir haben diese Beobachtung in unserem Algorithmus 5 berücksichtigt, indem wir nur $ggT(f_1, f_2)$ als multiplikative obere Schranke verwenden. Dadurch verkleinert sich die Bitgröße des ggT und wir benötigen weniger Primzahlen um ihn zu rekonstruieren. Für den Fall, dass D tatsächlich gebraucht wird, haben wir Wang's *Rational Reconstruction* wie folgt als *fallback* eingebaut. Um sicherzustellen, dass die Kosten der *Rational Reconstruction* nicht zu stark ins Gewicht fallen, arbeiten wir im Hybridansatz mit zwei Timern. Der Erste misst die Zeit, die der chinesischen Restsatz benötigt und der Zweite die Zeit von Wang's Algorithmus. Erst wenn die akkumulierten Zeiten des chinesischen Restsatzes größer sind als das Fünfzigfache des letzten *Rational Reconstruction* Algorithmus, wird dieser das nächste Mal aufgerufen.

Algorithm 5: **(Hybrid-Ansatz)** Given the polynomials F_1, $F_2 \in \mathbb{Z}(\alpha)[x]$ compute $G^\star \in \mathbb{Z}(\alpha)[x]$, the (up to constant factor) greatest common divisor of F_1 and F_2.

(1) Set $F_i = normalization\_factor(lc(F_i)) \cdot F_i$.
(2) Set $F_i = F_i / scalar\_factor(F_i)$.
(3) Set $f_i = lc(F_i) \in \mathbb{Z}$.
(3a) Set $D = denominator\_for\_algebraic\_integers(\mathbb{Z}(\alpha))$.
(3b) Set $\overline{g} = gcd(f_1, f_2)$ and $timer_{CR} = timer_W = 0$.
(4) Set $n = 0$, $e = min(deg(F_1), deg(F_2))$.
(5) Select p, a new odd prime not dividing f_1, f_2 or D.
(6) Set $\tilde{g} = \phi_p(\overline{g}) \in \mathbf{R}_p$, $\tilde{F}_i = \phi_p(F_i) \in \mathbf{R}_p[x]$.
(7) Invoke the euclidean algorithm to compute
$\tilde{G} = \tilde{g} \cdot gcd(\tilde{F}_1, \tilde{F}_2) \in \mathbf{R}_p[x]$.
(8) If $deg(\tilde{G}) = 0$, return $G^\star = 1$.
If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5).
If $deg(\tilde{G}) < e$, the former primes were unlucky.
Set $n = 0$, $e = deg(\tilde{G})$.
(9) Set $n = n + 1$.
(10) If $n = 1$, set the tuple $(q, G^\star) = (p, \tilde{G})$ and go back to step (5).
(11) Start $timer_{CR}$.
Use Chinese remainder to update the tuple (q, G^*):
$(q, G^\star) := chinese\_remainder((q, G^\star), (p, \tilde{G}))$.
Stop $timer_{CR}$.
(12) If the coefficients of $G^\star$ have changed go to step (15).
(13) Set $G^\star = G^\star / scalar\_factor(G^\star)$.
If $G^\star \nmid F_1$ or $G^\star \nmid F_2$ go back to step (15).
(14) return $G^\star$.
(15) If $timer_{CR} \leq 50 \cdot timer_W$ go back to step (5).
(16) Reset $timer_{CR}$ and $timer_W$.
(17) Start $timer_W$.
Try to use Wang's algorithm to compute
$(G^\star, d) := Wang(q, G^\star)$.
Stop $timer_W$.
(18) If Wang fails or $G^\star$ has changed go back to step (5).
(19) If $G^\star \nmid d \cdot F_1$ or $G^\star \nmid d \cdot F_2$ go back to step (5).
(20) return $G^\star$

Kapitel 4

Implementierung

Die vorausgegangenen Kapitel beschreiben die mathematischen Grundlagen und die Algorithmen zur Bestimmung des *ggT* zweier Polynome. In diesem Kapitel gehen wir auf die Implementierung der präsentierten Algorithmen als Teil der EXACUS Bibliothek [2] ein.

Ein wichtiges Ziel des EXACUS-Projekts ist es, die Wiederverwendbarkeit des Codes zu gewährleisten. Das bedeutet, die Implementierungen sollten so realisiert werden, dass die verwendeten Datentypen möglichst austauschbar sind, ohne den Code anpassen zu müssen. Aus diesem Grund folgt die Bibliothek und damit auch die von uns implementierten Algorithmen dem Paradigma der generischen Programmierung [1].

Abschnitt 4.1 gibt eine kurze Einführung in dieses Prinzip. Der zweite Teil dieses Kapitels stellt das EXACUS-Projekt vor und ordnet diese Arbeit in die Bibliothek ein. Abschnitt 4.3 befasst sich mit der konkreten Implementierung unserer Algorithmen. Dabei stellen wir die neu eingeführten Traits-Klassen vor, gehen auf das Nullteiler Problem in $\mathbf{R}_p = \mathbb{Z}/p\mathbb{Z}(\sqrt{r})$ ein und präsentieren eine erste Lösung dieses Problems. Zum Schluss erläutern wir Änderungen der Implementierung, die zu einer Laufzeitverbesserung führten.

4.1 Generische Programmierung

Die generische Programmierung ist zu einer wichtigen Technik in der Entwicklung wiederverwendbarer und effizienter Software-Bibliotheken geworden. Dabei werden Funktionen möglichst allgemein entworfen, um für unterschiedliche Datentypen und Datenstrukturen verwendet werden zu können. Das bedeutet, die Algorithmen werden nicht für einen bestimmten Datentyp geschrieben, sondern stellen nur gewisse Anforderungen an diese. Jazayeri *et. al* [17] definieren generische Programmierung folgendermaßen.

Definition 4.1 *Generische Programmierung ist ein Wissenszweig der Informatik, der versucht abstrakte Darstellungen von effizienten Algorithmen, Datenstrukturen und anderen Software Konzepten zu finden und diese zielbewusst zu organisieren. Das Ziel der generischen Programmierung ist Algorithmen und Datenstrukturen in einer allgemein wiederverwendbaren Form zu formulieren, die ihre unmittelbare Anwendbarkeit in der Softwareentwicklung ermöglichen. Die Schlüsselideen sind:*

- *Algorithmen mit möglichst wenig Annahmen an die Datentypen zu formulieren und umgekehrt um sie dadurch so kompatibel wie möglich zu machen.*
- *Anhebung eines konkreten Algorithmus auf eine möglichst allgemeine Ebene ohne an Effizienz zu verlieren. Das heißt, die abstrakteste Form ist derart, dass der für einen bestimmten Fall spezialisierte Algorithmus genauso effizient ist wie der Original Algorithmus.*
- *Wenn das Ergebnis der Anhebung nicht allgemein genug ist, um alle Anwendungen eines Algorithmus abzudecken, wird zusätzlich eine allgemeinere Form erstellt. Gleichzeitig wird aber auch sichergestellt, dass automatisch die effizienteste Spezialisierung gewählt wird, die anwendbar ist.*
- *Es ist möglich, mehr als einen generischen Algorithmus für denselben Zweck und von gleichem Abstraktionsgrad bereitzustellen, falls kein Algorithmus einen der anderen an Effizienz für die Gesamtheit der Eingaben dominiert. Dies erfordert eine ausreichend genaue Beschreibung der Eingabebereiche, für die ein Algorithmus der effizienteste ist.*

Im Folgenden wollen wir auf die für unsere Aufgabe wichtigen Aspekte der generischen Programmierung eingehen.

4.1.1 Templates

Die im EXACUS-Projekt verwendete Programmiersprache C++ ermöglicht diese vom Datentyp unabhängige Programmierung durch die bereitgestellten Konzepte der *Funktions- und Klassentemplates*. Templates sind unvollständig definierte Objekte, mit der Besonderheit, dass einige Datentypen nur durch einen Platzhalter kenntlich gemacht werden. Diese Platzhalter nennt man *Template-Argumente*. Der Templatecode verhält sich wie eine Schablone: Zur Compilezeit werden die Template-Argumente durch konkrete Datentypen ersetzt. Hierbei werden beispielsweise für alle Aufrufe einer Templatefunktion mit unterschiedlichen Parametertypen jeweils eigene Varianten, mit den entsprechenden Typen, erzeugt und kompiliert. Dies führt in der Regel zu längeren Compilezeiten, da sich der ursprüngliche Templatecode nicht vorab compilieren lässt. Ein Funktionstemplate verhält sich also wie eine Funktion, die in der Lage ist, Argumente verschiedener Typen entgegenzunehmen und verschiedene Rückgabetypen zu liefern. Ein Klassentemplate wendet das gleiche Prinzip auf Klassen an.

4.1.2 Konzepte und Modelle

Natürlich ist nicht jeder beliebige Datentyp als Template-Argument eines bestimmten Funktions- oder Klassentemplates zulässig. Jeder generische Algorithmus stellt gewisse Anforderungen, die erfüllt werden müssen, damit der Datentyp als Argument zulässig ist. Ein Algorithmus sollte dabei möglichst wenig vom Datentyp fordern. Die C++ Standard Template Library (STL) [1] teilt diese Anforderungen in sogenannte *Konzepte* ein. Ein Datentyp, der diese Anforderungen erfüllt, wird als ein *Modell* dieses Konzeptes bezeichnet. Ein Konzept, welches ein anderes Konzept verfeinert, nennt man *Verfeinerung*.

Als Beispiel betrachten wir die algebraischen Strukturen, die wir in Abschnitt 2.1 eingeführt haben. In EXACUS werden zum Beispiel die Eigenschaften eines faktoriellen Ringes und eines euklidischen Ringes zu den Konzepten `UFDomain` und `EuclideanRing` zusammengefasst, wobei der `EuclideanRing` eine Verfeinerung

des Konzeptes `UFDomain` darstellt. Modelle des Konzeptes `EuclideanRing` sind zum Beispiel `leda::Integer` [23] oder `CORE::BigInt` [18]. Für eine genaue Definition der Konzepte verweisen wir auf das Bedienungshandbuch des CGAL Paketes *Algebraic Foundations* [14].

4.1.3 Traits-Klassen, Tags und Funktoren

Oft ist es wichtig, während man ein Funktions- oder Klassentemplate schreibt, zusätzliche Informationen über das Template-Argument zu kennen. Je nach Eigenschaft des Datentyps, soll zum Beispiel ein anderes Verhalten gezeigt werden, wobei man im Allgemeinen ein grundlegendes Verhalten spezifiziert und dann nach Bedarf für spezielle Datentypen eine eigene Behandlung definiert. Der Trick ist, sich eine Hilfsklasse zunutze zu machen, die *Traits-Klasse*. Traits ist englisch für Eigenschaften und meint die Eigenschaften der Klasse abhängig von ihrem Datentyp. Nehmen wir als Beispiel die von der STL bereitgestellte Traits-Klasse `std::iterator_traits<T>`, die in etwa so aussieht:

```
template < class Iterator >
struct iterator_traits {
    typedef ... iterator_category ;
    typedef ... value_type ;
    typedef ... difference_type ;
    typedef ... pointer ;
    typedef ... reference ;
};
```

Diese Traits-Klasse ist für bestimmte Iteratoren spezialisiert. Abhängig vom Iterator–Typ `Iterator`, mit dem die Klasse instantiiert wird, gibt `iterator_category()` ein anderes Objekt zurück, zum Beispiel `forward_iterator` oder `random_iterator`. Diese Eigenschaften, die das Template-Argument kennzeichnen, werden auch *Tags* genannt. Oft wird ein Tag als Argument an eine überladene Funktion übergeben, um so den besten Algorithmus für das Template-Argument zu finden. Ein weiteres Beispiel ist die Klasse `Polynomial`, auf die wir unter Abschnitt 4.2 kurz eingehen werden.

Auch für bestimmte Algorithmen existieren Traits-Klassen, zum Beispiel `Chinese_remainder_traits` und `Wang_traits`, die wir in Abschnitt 4.3 näher beschreiben werden. Diese stellen Funktionalitäten bereit, die vom jeweilige Algorithmus benötigt werden. Die Algorithmen kommunizieren dann über die zugehörigen, spezialisierten Traits-Klassen mit den entsprechenden Datentypen. Möchte man einen vorhandenen Algorithmus auf einen neuen Datentyp anwenden, muss man nur die betreffende Traits-Klasse für diesen Typ spezialisieren.

Funktionsobjekte oder *Funktoren* sind Objekte, die genauso aufgerufen werden können wie Funktionen. Im Allgemeinen ist es ein Objekt einer Klasse, welches den Operator () definiert. Funktoren verhalten sich wie Funktionen, haben aber trotzdem alle Eigenschaften von Objekten.

Insgesamt kann man sagen, generische Programmierung erhöht zwar die Komplexität einer Aufgabe zum Teil enorm, da Code unter verschiedensten Bedingungen benutzt und konfiguriert werden kann, oft erweist sie sich aber als mächtiges Mittel, um Code Redundanzen zu vermeiden, die Performance hoch zu halten und die Wartbarkeit einer Software zu verbessern.

4.2 Das EXACUS Projekt

Das EXACUS-Projekt (*Efficient and Exact Algorithms for Curves and Surfaces* [1]) ist eine Sammlung von C++ Bibliotheken, die vom Max-Planck-Institut für Informatik in Saarbrücken gegründet wurden. Das Projekt beinhaltet effiziente, exakte und vollständige Algorithmen, um geometrische Probleme für Objekte von höherem Grad zu lösen. Die Bibliotheken berechnen zum Beispiel Anordnungen von Kegeln und Quadriken (Algebraische Flächen vom Grad 2) in einer Ebene und implementieren unter anderem die mathematischen Grundlagen, die wir in Kapitel 2 eingeführt haben.
Die Algorithmen sind exakt und vollständig, da sie für jede sinnvolle Eingabe,

[1] `http://www.mpi-inf.mpg.de/projects/EXACUS/`

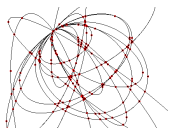
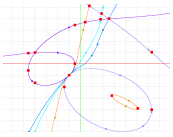
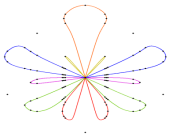
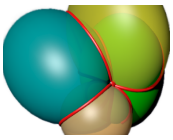
<table>
<tr><td colspan="2"> 
CONIX (CnX)</td><td colspan="2"> 
CUBIX (CbX)</td><td colspan="2"> 
ALCIX (AcX)</td><td colspan="2"> 
QUADRIX (QdX)</td></tr>
<tr><td colspan="8">SWEEPX (SoX)</td></tr>
<tr><td colspan="8">NUMERIX (NiX)</td></tr>
<tr><td colspan="8">Library Support (LiS)</td></tr>
<tr><td>GMP</td><td>CORE</td><td>LEDA</td><td>NTL</td><td>BOOST</td><td>CGAL</td><td>QT</td><td>QI</td></tr>
</table>

Abbildung 4.1: Aufbau der Bibliotheken des EXACUS-Projekts.

einschließlich der degenerierten Fälle, das mathematisch richtige Ergebnis berechnen. Sie sind effizient gemessen an ihren Laufzeiten.

Die EXACUS Bibliotheken sind wie in Abbildung 4.1 geschichtet angeordnet. Die Grundlage bilden externe Bibliotheken, wobei das Herzstück der EXACUS-Funktionalität, also die mathematischen Berechnungen, unabhängig davon bleiben. EXACUS benutzt beispielsweise die Zahlentypen aus LEDA und CORE, sowie die Intervallarithmetik von BOOST. Die Spitze bilden die vier Bibliotheken: CONIX, QUADRIX, ALCIX und CUBIX. Für eine genauere Beschreibung verweisen wir auf die jeweiligen Arbeiten [3], [4] , [9] und [10].

Im Zusammenhang mit dieser Diplomarbeit spielt die Bibliothek NUMERIX eine wichtige Rolle. Hier werden mathematische Grundlagen, wie die Konzepte der Zahlentypen, implementiert und numerische Algorithmen, beispielsweise unsere *ggT* Berechnung, bereitgestellt. Zudem werden Hilfsmittel entwickelt, um aus einem vorhandenen Datentyp einen neuen Datentyp zu konstruieren. Betrachten wir als Beispiel Polynome über einem generischen Datentyp:

Die Klasse Polynomial<AS>

Die Klasse `Polynomial<AS>` beschreibt Polynome über dem Datentyp `AS`. Abhängig von den Eigenschaften des Zahlentyps, die in der zugehörigen Traits-

Klasse `Algebraic_structure_traits<AS>` gespeichert sind, passt sich die Polynomklasse an und wählt die besten Implementierungen für gewisse Funktionen aus. Zum Beispiel wird der `Algebra_type` von `Polynomial<AS>` durch den `Algebra_type` von `AS` bestimmt. Erfüllt `AS` das Konzept `EuclideanRing`, so ist der `Algebra_type` von `Polynomial<AS>` `UFDomain`. Erfüllt der Koeffizientyp das Konzept `Field`, so ist `Polynomial<AS>` ein `EuclideanRing`. Der `Algebra_type` der Polynomklasse wird nun als Argument an bestimmte überladene Funktionen übergeben, die sich entsprechend anpassen. Nehmen wir als Beispiel die Funktion zur *ggT*–Berechnung. Ist `Polynomial<AS>` ein Konzept des Modells `EuclideanRing`, so wird der *ggT* mit Hilfe des euklidischen Algorithmus berechnet. Ist der `Algebra_type` der Polynomklasse `UFDomain` und ist es möglich die modularen Bilder der Koeffizienten zu bestimmen, so wird zur *ggT*-Berechnung der Algorithmus von Brown aufgerufen. Stammen die Koeffizienten aus einer algebraischen Erweiterung von $\mathbb{Z}$, so ist der `Algebra_type` von `Polynomial<AS>` `IntegralDomain` (Integritätsbereich). Wenn wir auch hier die modularen Bilder der ganzzahligen Koeffizienten bestimmen können, rufen wir unseren Hybrid-Algorithmus auf. Ob die modularen Bilder der Koeffizienten bestimmbar sind sagt uns die Funktion `Is_modularizable()` der *Traits-Klasse* `Modular_traits`.

4.3 Implementierung der Algorithmen

4.3.1 Traits-Klassen

Wir haben verschiedene Traits-Klassen eingeführt, um die Funktionalitäten bereitzustellen, welche die Algorithmen aus Kapitel 3 benötigen. Wir gehen im Folgenden kurz auf diese Klassen ein. Für eine detailliertere Beschreibung verweisen wir auf den Anhang von [13].

Scalar Factor Traits

Die Traits-Klasse `Scalar_factor_traits` wird benutzt, um überflüssige Skalare aus Polynomen über $\mathbb{Z}(\sqrt{r})$ zu entfernen, vergleiche Abschnitt 3.2.2 und Algorithmus 2 und 4. Da für Zahlen aus $\mathbb{Z}(\sqrt{r})$ der *ggT* nicht definiert ist, stellt uns die

Traits-Klasse den Funktor `Scalar_factor` zur Verfügung, der den skalaren Faktor solcher Zahlen berechnet. Für einen Koeffizienten $a+b\sqrt{r} \in \mathbb{Z}(\sqrt{r})$ ist dieser definiert als `Scalar_factor`$(a+b\sqrt{r}) = ggT\,(a,b)$. Nun lässt sich der überflüssige Faktor des Polynoms als ggT der skalaren Faktoren der einzelnen Koeffizienten berechnen.

Die Klasse ist darüber hinaus von allgemeinem Interesse. Überflüssige Skalare aus einem zusammengesetzten Datentyp zu entfernen, obwohl der Koeffiziententyp keinen ggT bereitstellt, taucht als Problem immer wieder auf. Die Klasse `Scalar_factor_traits` löst dieses Problem zum Beispiel noch für Vektoren und Matrizen.

Da wir den ggT benutzen, ist die Klasse `Scalar_Factor_Traits` nur anwendbar, wenn der innerste Koeffiziententyp ein faktorieller Ring ist.

Algebraic Extension Traits

Die Traits-Klasse `Algebraic_extension_traits` liefert Funktionalitäten in Zusammenhang mit algebraische Erweiterungen und stellt insbesondere die beiden Funktoren `Normalization_factor` und `Denominator_for_algebraic_integer` zur Verfügung.

- `Normalization_factor(x)`
  Für ein gegebenes $x \in \mathbb{Z}(\alpha)$ berechnet `Normalization_factor` einen Faktor $a \in \mathbb{Z}(\alpha)$, so dass $x \cdot a \in \mathbb{Z}$. Dieser Funktor wird in den Algorithmen 2 und 4 benutzt, um die gegebenen Polynome so zu manipulieren, dass ihr Leitkoeffizient aus $\mathbb{Z}$ ist.

- `Denominator_for_algebraic_integer`
  Dieser Funktor berechnet die multiplikative Schranke der Nenner von rationalen Zahlen, die im ggT von Polynomen entstehen können, obwohl die Koeffizienten der Polynome aus einer ganzen algebraischen Erweiterung von $\mathbb{Z}$ stammen (vgl. Abschnitt 3.2.3).

Chinese Remainder Traits

Die Traits-Klasse `Chinese_remainder_traits` ermöglicht den Zugriff auf den chinesischen Restsatz. Sie ist so konzipiert, dass der chinesische Restsatz auch auf zusammengesetzte Datentypen anwendbar ist, die in einzelne ganzzahlige Koeffizienten zerlegt werden können. Zum Beispiel ruft der chinesische Restsatz für Polynome den chinesischen Restsatz für jeden einzelnen Koeffizienten auf und liefert das resultierende Polynom zurück. Der chinesische Restsatz, angewendet auf Elemente $a_1 + b_1\sqrt{r_1}$ und $a_2 + b_2\sqrt{r_2}$, berechnet den chinesischen Restsatz für die Koeffizienten $(a_1, a_2), (b_1, b_2), (r_1, r_2)$ und liefert ein Ergebnis $a_3 + b_3\sqrt{r_3}$ zurück ($a_i, b_i, r_i \in \mathbb{Z}$).

Wie im Beweis des chinesischen Restsatzes 2.28 gezeigt, benötigen wir den erweiterten euklidischen Algorithmus, um das Ergebnis zu berechnen. Deshalb ist die Klasse `Chinese_remainder_traits` nur anwendbar, wenn der innerste Koeffizientyp ein euklidischer Ring ist.

Wang Traits

Die Traits-Klasse `Wang_traits` ermöglicht den Zugriff auf den *Rational Reconstruction* Algorithmus von Wang (Algorithmus 3). Die Gestalt der Traits-Klasse ist symmetrisch zur `Chinese_remainder_traits`. Der Algorithmus für Polynome zum Beispiel ruft wieder Wangs Algorithmus für jeden einzelnen Koeffizienten auf und liefert den Hauptnenner aller Einzelergebnisse und das entsprechende Zählerpolynom zurück.

4.3.2 Das Nullteiler-Problem in $\mathbf{R}_p$

Wie unter Abschnitt 3.2.2 schon kurz erwähnt, funktioniert die *ggT*-Berechnung für Polynome über einer algebraischen Erweiterung von $\mathbb{Z}$ nicht immer. Das Problem ist, dass der modulare Koeffiziententyp $\mathbf{R}_p = \mathbb{Z}/p\mathbb{Z}(\sqrt{r})$ nicht nullteilerfrei ist.
Für $2+3\sqrt{2} \in \mathbb{Z}/7\mathbb{Z}(\sqrt{2})$ gilt zum Beispiel

$$(2+3\sqrt{2}) \cdot (2-3\sqrt{2}) = -14 \equiv 0 \mod 7,$$

damit ist $2+3\sqrt{2}$ ein Nullteiler in $\mathbf{R}_7$. Es ist also nicht möglich mit einem Polynom zu dividieren dessen Leitkoeffizient ein Nullteiler ist. Weder Langemyr und McCallum [22] noch Encarnacion [11] sind auf dieses Problem in ihren Arbeiten weiter eingegangen.

In der EXACUS-Bibliothek wird beim Anlegen einer Instanz der Klasse `Polynomial` mit Hilfe der Funktion `is_zero` geprüft, ob der Leitkoeffizient Null oder ein Nullteiler ist. Ist dies der Fall, wird der Leitkoeffizient auf Null gesetzt und das Polynom hat einen kleineren Grad. Insbesondere verschwinden also auch alle Leitkoeffizienten die Nullteiler sind. So wird vermieden, dass man versucht mit einem Nullteiler zu dividieren. Allerdings entstehen hieraus weitere Probleme, die uns erst im Laufe der Arbeit bewusst wurden.
Nach Lemma 3.2 konnten wir für Polynome über $\mathbb{Z}$ annehmen, dass der modulare ggT höchstens einen zu großen Grad annimmt. Unglückliche Primzahlen wurden demnach nur über den Grad des modular berechneten ggT identifiziert. Für Polynome über algebraischen Erweiterungen ist dieses Verfahren leider nicht mehr ausreichend, da eine weitere Kategorie von unglücklichen Primzahlen hinzukommt. Wir müssen zusätzlich beachten, dass der Leitkoeffizient eines modularen Polynoms in der polynomiellen Restesequenz (PRS) ein Nullteiler sein kann. Dieser wird dann wie erwähnt verworfen. Demzufolge ändern sich natürlich auch die weiteren Polynome der Restefolge und schließlich der modulare ggT $\tilde{G}$. Insbesondere kann der Grad von $\tilde{G}$ nun auch kleiner sein als der des korrekten ggT.

Nehmen wir an, G_p sei ein korrekt berechneter modularer ggT vom richtigen Grad und G_q der modulare ggT bezüglich einer unglücklichen Primzahl q. Folgende Situationen können eintreten und verdeutlichen, dass es nicht mehr ausreicht unglückliche Primzahlen über den Grad des berechneten ggT zu ermitteln:

1. $grad(G_q) > grad(G_p)$:
   Die Primzahl q wird richtigerweise als unglücklich erkannt und das Ergebnis G_q wird verworfen.

2. $grad(G_q) < grad(G_p)$:
   Nach der Vorschrift des Algorithmus verwerfen wir das richtige Ergebnis G_p und speichern den falschen ggT G_q. Die glücklichen Primzahlen berech-

nen nun aber einen ggT, dessen Grad größer ist als $grad(G_q)$. Das bedeutet, wir werden alle weiteren richtigen Ergebnisse verwerfen und der Algorithmus wird nicht terminieren.

3. $grad(G_q) = grad(G_p)$, aber die PRS hat sich geändert:
   Der modulare ggT G_q wird in die Berechnungen des chinesischen Restsatzes mit aufgenommen. Das bedeutet, der korrekte ggT kann nicht gefunden werden, da ein falsches Zwischenergebnis in die Rechnung mit einfließt.

Aus diesem Grund speichern wir in einem sogenannten *Gradvektor* die Grade aller Polynome der PRS.

Gradvektor der PRS

Der erste Eintrag des Vektors ist der Grad des ersten Restpolynoms, der Vorletzte der Grad des ggT und der letzte Eintrag eine Null. Für solch einen Gradvektor v mit n Einträgen gilt also:

$$v[0] > v[1] > \ldots > v[n-2] = Grad(ggT) \geq v[n-1] = 0.$$

Wir definieren weiterhin die lexikographische Ordnung der Gradvektoren $v = (v_0, \ldots, v_n)$ und $w = (w_0, \ldots, w_m)$ wie folgt. Ist $n = m$ und $v_i = w_i$ für $i = 0, \ldots, n$, dann gilt $v = w$. Andererseits sei j die kleinste Zahl für die $v_j \neq w_j$. Ist $v_j < w_j$, so gilt $v < w$ und umgekehrt.

Schauen wir uns zuerst noch einmal an, wie wir bisher in den Algorithmen 2, 4 und 5 die unglücklichen Primzahlen verworfen haben, ohne auf das Problem der Nullteiler in $\mathbf{R}_p$ einzugehen.

(7) Invoke the euclidean algorithm to compute
    $\tilde{G} = \tilde{g} \cdot gcd(\tilde{F}_1, \tilde{F}_2) \in \mathbf{R}_p[x]$.
(8) If $deg(\tilde{G}) = 0$, return $G^\star = 1$.
    If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5).
    If $deg(\tilde{G}) < e$, the former primes were unlucky, set n = 0, e = deg($\tilde{G}$).
(9) Set $n = n + 1$.

Nun können wir mit Hilfe der Gradvektoren und der zugehörigen Ordnungsrelation eine Veränderung der PRS feststellen und die jeweilige Primzahl als unglücklich erkennen und verwerfen. Dazu speichern wir den Gradvektor der alten PRS in `prs_degree_old` und den neuen Gradvektor in `prs_degree_new`. Die Algorithmen haben wir folgendermaßen angepasst, wobei für die erste Primzahl `prs_degree_old` = `prs_degree_new` gesetzt wird.

(7) Invoke the euclidean algorithm to compute
$\tilde{G} = (\tilde{g}) \cdot gcd(\tilde{F}_1, \tilde{F}_2) \in \mathbf{R}_p[x]$, and save the degrees of the PRS
in `prs_degree_new`.

(8) If `prs_degree_new` $<$ `prs_degree_old`, p is an unlucky prime,
so go back to step (5).
If `prs_degree_new` $>$ `prs_degree_old`, the former primes were unlucky,
set $n = 0$ and `prs_degree_old` = `prs_degree_new`.

(8a) If $deg(\tilde{G}) = 0$, return $G^\star = 1$.

(9) Set $n = n + 1$.

Gehen wir an einem Beispiel die Fälle der unglücklichen Primzahlen durch, um zu zeigen, dass dieses Vorgehen korrekt ist. Nehmen wir an, die korrekten Grade der PRS sind `prs_degree` $= (6,5,4,3,0)$. Das bedeutet der gesuchte ggT hat den Grad 3.

1. In `prs_degree_old` sind die korrekten Grade der PRS gespeichert. Die neue Primzahl p ist unglücklich und vergrößert den Grad des ggT, zum Beispiel gilt `prs_degree_new` $= (6,5,4,0)$. Dann folgt

   $$\texttt{prs\_degree\_new} < \texttt{prs\_degree\_old}$$

   und wir verwerfen die Primzahl p.

2. Die alten Primzahlen waren unglücklich und berechnen einen zu großen ggT (zum Beispiel `prs_degree_old` $= (6,5,4,0)$). Die neue Primzahl berechnet die richtige PRS. Dann gilt

   $$\texttt{prs\_degree\_new} > \texttt{prs\_degree\_old}$$

   und wir verwerfen alle alten Primzahlen.

3. Durch die neue Primzahl p wird ein Leitkoeffizient in der PRS zu einem Nullteiler. Die Grade in der polynomiellen Restefolge ändern sich und wir erhalten einen ggT von kleinerem Grad (zum Beispiel einen konstanten ggT `prs_degree_new` $= (5,4,3,2,1,0,0)$). In `prs_degree_old` sind die korrekten Grade gespeichert. Es gilt also

   $$\texttt{prs\_degree\_new} < \texttt{prs\_degree\_old}$$

   und die neue Primzahl p wird verworfen.

4. Durch die alten Primzahlen wurde ein Leitkoeffizient in der polynomiellen Restefolge zu einem Nullteiler, wodurch sich die Grade ändern (zum Beispiel `prs_degree_old` $= (5,4,3,0)$). Die neue Primzahl berechnet die richtige PRS. Dann gilt

   $$\texttt{prs\_degree\_new} > \texttt{prs\_degree\_old}$$

   und wir verwerfen alle alten Primzahlen.

Um auszuschließen, dass die beiden Polynome nur teilerfremd sind, weil ein Nullteiler in der PRS aufgetaucht ist, wird der Test $Grad(\tilde{G}) = 0$ erst in Schritt (8a) durchgeführt.

Dieses ist ein erster naiver Ansatz, um das Problem der Nullteiler in $\mathbf{R}_p$ in den Griff zu bekommen. Ein Problem besteht noch, wenn durch die erste Primzahl ein Nullteiler in der PRS auftritt und der dadurch berechnete modulare ggT konstant ist. Wir würden also fälschlicherweise folgern, dass die beiden Polynome teilerfremd sind.
Wir könnten dieses Ergebnis durch eine weitere Rechnung bestätigen lassen, um die ohnehin sehr geringe Wahrscheinlichkeit für diesen Fall weiter zu minimieren. Ein Grundsatz des EXACUS-Projekts ist es aber für alle Fälle das richtige Ergebnis zu berechnen, welcher somit verletzt würde. Aus diesem Grund ist ein anderer Lösungsansatz angedacht, der diese Maxime erfüllt und einen saubereren Programmierstil darstellt. Dieser Ansatz wird unter Kapitel 6 kurz vorgestellt.

4.3.3 Verbesserung der Implementierung

Im Laufe der Tests und Benchmarks haben wir einige Änderungen an den Algorithmen vorgenommen, mit dem Ziel die Laufzeit zu verbessern. Zwei besonders wirkungsvolle Verbesserungen wollen wir im Folgenden vorstellen. Vergleichen Sie hierzu auch die Auswirkungen auf die Laufzeiten, die wir unter Abschnitt 5.2 diskutieren werden.

Der Divides – Funktor

Wir können die Algorithmen, wie unter Abschnitt 3.1.2 besprochen, grob in vier Phasen einteilen: Initialisierungsphase, modulare Phase, Rekonstruktionsphase und Verifikationsphase. In der Verifikationsphase wurde zu Beginn mit Hilfe der Pseudo-Division (Lemma 2.18) geprüft, ob der berechnete $ggT\ G^\star$ ein Teiler der beiden ursprünglichen Polynome ist. Die Pseudo-Division ist zwar über jedem Integritätsbereich anwendbar, dafür aber wie allgemein bekannt eine äußerst teure Operation (vergleiche zum Beispiel [19]). Zusätzlich bestimmt dieser Algorithmus den Rest der Division. Da wir aber nur daran interessiert sind, ob die Division aufgeht, stellt das für uns einen unnötigen Aufwand dar.

Aus diesem Grund haben wir den Funktor `Divides` entwickelt, der noch in die Klasse `Algebraic_structure_traits` eingebaut werden muss. Diese Traits–Klasse und ihre Funktionalitäten werden in Kapitel 2.2 der Dissertation von Michael Hemmer [15] vorgestellt. Schauen wir uns zuerst an, wie der `Divides` Funktor in einer beliebigen Testfunktion benutzt wird.

```
template<class AS>
bool test_function(const AS& x, const AS& y){
    typedef CGAL::Algebraic_structure_traits< AS >   AST;
    typename AST::Divides divides;
    AS q;
    bool result = divides(x,y,q);
    return result;
}
```

In der ersten Zeile der templatisierten Funktion wird für die Spezialisierung der

Traits-Klasse `Algebraic_structure_traits` für den Datentyp AS der kürzere Name AST eingeführt. Danach wird ein Objekt des Funktors `Divides` instantiiert. Das Objekt q vom Typ AS dient als Rückgabewert des Funktors. In der vierten Zeile benutzen wir den `Divides` Funktor, der eine Boolesche Variable liefert, je nachdem, ob „x teilt y"gilt oder nicht. Wird „*true*"zurückgeliefert, gilt ausserdem $x \cdot q = y$.

Gehen wir nun auf die Implementierung des Funktors ein. Um wieder den besten Algorithmus für das jeweilige Template-Argument zu finden, bestimmt der `Algebra_type` von AS wie der `Divides` Funktor vorgeht. Für folgende Datentypen AS wurde eine Spezialisierung implementiert (Eine Erläuterung der Funktoren `Integral_division` und `Div` ist in [14] zu finden):

- `UFDomain`:
  Hier wird getestet, ob $ggT(x,y) = unit\_normal(x)$ gilt. Wenn ja, setzen wir $q = Integral\_division(y,x)$ und geben „*true*"zurück, ansonsten „*false*".

- `EuclideanRing`:
  Hier wird $q = Div(y,x)$ berechnet. Falls $q \cdot x = y$ gilt, geben wir „*true*" zurück, andernfalls „*false*".

- `Field`:
  Die Division in einem Körper funktioniert für $x \neq 0$ immer. Demnach wird $q = y/x$ gesetzt und „*true*"zurückgeliefert.

- `IntegralDomain (Sqrt_extension<COEFF, COEFF>)`:
  Dieser zusammengesetzte Datentyp ist ein Integritätsbereich, es existiert also keine Division mit der wir den Test durchführen können. Aus diesem Grund zerlegen wir den Teilbarkeitstest und benutzen den `Divides` Funktor des Datentyps COEFF. Überlegen wir zuerst, was es bedeutet eine solche Zahl zu invertieren. Für $x = x_0 + x_1\sqrt{r}$ bezeichnen wir mit $\overline{x} = x_0 - x_1\sqrt{r}$ das algebraisch Konjugierte zu x und es gilt $x \cdot \overline{x} = x_0^2 - x_1^2 r$. Somit folgt $x^{-1} = \overline{x}/(x \cdot \overline{x}) = \overline{x}/(x_0^2 - x_1^2 r)$ und damit

$$\frac{y}{x} = \frac{y \cdot \overline{x}}{x_0^2 - x_1^2 r}. \tag{4.1}$$

Um nun zu testen, ob diese Division aufgeht, berechnen wir den Nenner $n = n_0 + n_1 r = x_0^2 - x_1^2 r \in \mathbb{N}$ und den Zähler $z = z_0 + z_1\sqrt{r} = y \cdot \bar{x}$ aus (4.1). Dann prüfen wir mit Hilfe des `Divides` Funktors von COEFF, ob der Nenner die skalaren Faktoren des Zählers dividiert (`divides`(n, z_0, q_0), `divides`(n, z_1, q_1)). Ist einer der beiden Tests falsch, geben wir „*false*“zurück, andernfalls setzen wir $q = q_0 + q_1\sqrt{r}$. Gilt dann $q \cdot x = y$, folgt „x teilt y“und wir geben „*true*“zurück.

- `Polynomial<COEFF>`:
  Für COEFF vom `Algebra_type` `IntegralDomain, UFDomain` und `EuclideanRing` wurde eine Polynomdivision mit Hilfe der eben beschriebenen `Divides` Funktoren implementiert. Die Polynomdivision bricht ab und der Funktor liefert „*false*“zurück, sobald eine Division nicht aufgeht. Sind alle Divisionen korrekt, wird geprüft, ob der Rest der Polynomdivision Null ist. Wenn ja, liefert der Funktor „*true*“zurück.
  Für COEFF vom Typ `Field` bilden die Polynome einen euklidischen Ring. Hier ist es am effizientesten, den euklidischen Algorithmus aufzurufen und dann zu testen, ob das Restpolynom Null ist.

Cache für den erweiterten euklidischen Algorithmus

Wie wir im Beweis des Chinesischen Restsatzes 2.28 gesehen haben, benötigen wir den erweiterten euklidischen Algorithmus um für unsere teilerfremden Moduli m_p (die neue Primzahl) und m_{pq} (das Produkt der alten Primzahlen) zwei Zahlen s und t zu bestimmen, so dass $sm_p + tm_{pq} = 1$ gilt. Werden in einem Programm nacheinander verschiedene ggT-Tests durchgeführt, starten wir jedesmal mit derselben Primzahl. Wie bereits erwähnt, sind unglückliche Primzahlen sehr selten und somit bleibt die Sequenz der verwendeten Primzahlen in der Regel unverändert. Der erweiterte euklidische Algorithmus würde demnach in einem Programmdurchlauf mehrmals dieselbe Rechnung ausführen. Daher haben wir einen *cache* (engl. für Speicher) für bereits berechnete Moduli-Paare eingebaut. Das heißt, wird der erweiterte euklidische Algorithmus für zwei Moduli aufgerufen, für die s und t bereits berechnet wurden, werden die zugehörigen Werte aus dem *cache* abgerufen, ohne den Algorithmus ein weiteres Mal auszuführen.

Kapitel 5

Benchmarks

Um die Erkenntnisse der theoretischen Komplexitätsanalysen der Abschnitte 3.1.3 und 3.2.5 zu belegen und den praktischen Nutzen unserer Ansätze zu demonstrieren, schauen wir uns die Laufzeiten der Algorithmen für verschiedene Instanzen von Polynompaaren an. Zur besseren Bewertung dieser Ergebnisse wurden die gleichen Berechnungen mit Hilfe der Computer Algebra Systeme NTL [1] und SINGULAR [2] durchgeführt.

Die Benchmarks wurden auf einem Pentium(R) M Prozessor mit 1.7 GHz und 512 KB Cache unter Linux erstellt. Die ausführbaren Programme wurden mit Hilfe des GNU C++ Compilers Version 3.4.6, der Optimierung (`-O3`) und deaktivierter Fehlersuche (`-DNDEBUG`) kompiliert. Um genauere Laufzeiten der Algorithmen zu erhalten, wurde jede Instanz von Polynompaaren für mindestens 50 Sekunden berechnet und die resultierende Zeit durch die Anzahl der Durchläufe dividiert.

Im ersten Abschnitt dieses Kapitels stellen wir die verschiedenen Polynominstanzen vor und gehen kurz auf die Computer Algebra Systeme NTL und SINGULAR ein. Im zweiten Teil zeigen wir an einem Beispiel, wie sich die unter 4.3.3 vorgestellten Verbesserungen der Implementierung auf die Laufzeiten auswirken. In den Abschnitten 5.3 und 5.4 vergleichen wir die Laufzeiten der verschiedenen Algorithmen zur Berechnung des ggT für Polynome über $\mathbb{Z}$ bzw. über einer algebraischen Erweiterung von $\mathbb{Z}$ vom Grad 2.

[1] `http://www.shoup.net/ntl/`

5.1 Allgemeine Bemerkungen

5.1.1 Die Testpolynome

Wir betrachten drei Kenngrößen, um die Entwicklung der Laufzeiten zu beschreiben: die Bitlänge der Koeffizienten, den Polynomgrad und den Grad des *ggT*. Um die Bedeutung einer der drei Kenngrößen zu untersuchen, verändern wir dessen Größe und halten die beiden anderen Parameter konstant. Um die Genauigkeit der Ergebnisse zu erhöhen, wurden für jede Einstellung 50 zufällige Polynompaare generiert. Jedes Polynompaar besteht aus dem *ggT* und den teilerfremden Kofaktoren, jeweils vom gewünschtem Grad und gewünschter Koeffizientengröße. Diese drei Faktoren sind zufällig, da jeder ganzzahlige Koeffizient gewürfelt wird, so dass er die gewünschte Bitlänge besitzt. Das führende Bit wurde jeweils auf 1 gesetzt.

Um die Bedeutung der Bitlänge der Koeffizienten zu analysieren, haben wir zwei Familien von Dateien mit univariaten Polynomen erzeugt. Eine Familie mit ganzzahligen Koeffizienten und eine mit Koeffizienten aus einer algebraischen Erweiterung vom Grad 2. Hier unterscheiden wir zusätzlich, ob sich die Bitlängen der Koeffizienten des *ggT* und der Kofaktoren gemeinsam ändern oder nur eine Größe ansteigt.

Da der Grad des *ggT* Einfluss auf die Länge der polynomiellen Restefolge (PRS) hat und damit auf die Laufzeit der Algorithmen, haben wir zwei weitere Dateienfamilien mit Polynompaaren erzeugt: Wieder eine Familie mit ganzzahligen Koeffizienten und eine mit Koeffizienten aus einer algebraischen Erweiterung der ganzen Zahlen vom Grad 2. Eine Datei enthält 50 zufällige Polynompaare vom gewünschten Grad d und von gewünschter Bitlänge der Koeffizienten. Diesmal unterscheiden sich die Dateien darin, dass der Grad des *ggT* variiert $(1, 2, \ldots, d-1)$.

Mit der dritten Dateienfamilie untersuchen wir den Einfluss des Polynomgrades auf die Laufzeit. Dazu halten wir die Bitlänge der Koeffizienten und den Grad des *ggT* konstant und erhöhen den Grad der Kofaktoren.

Unter `NumeriX/include/NiX` wurde von uns die Datei `gen_polynomials.h` implementiert, die verschiedene Funktionen zur Verfügung stellt, um zufällige Polynome zu generieren. Mit Hilfe dieser Funktionen konnten wir die benötigten Polynompaare wie gewünscht erzeugen.

5.1.2 NTL und SINGULAR

NTL ist eine hochqualitative, leistungsstarke C++ Bibliothek, die von Victor Shoup entwickelt wurde. Sie stellt unter anderem Algorithmen für beliebig große ganze Zahlen und Polynome über den ganzen Zahlen zur Verfügung. Insbesondere die Polynomarithmetik ist eine der schnellsten überhaupt und wird vom Autor explizit gelobt. Allerdings ist die NTL-Bibliothek nur für Polynome über dem internen Datentyp geschrieben und nicht, wie EXACUS, generisch programmiert. Das bedeutet, wir müssen unsere Polynome, die über dem Zahlentyp CORE definiert sind, in den NTL Zahlentyp transformieren. Um die reine NTL Zeit ohne Transformationskosten und die Zeit des NTL-Algorithmus, aufgerufen aus EXACUS, unterscheiden zu können, haben wir beide Zeiten getrennt geplottet. Da die NTL-Bibliothek keinen Zahlentyp für algebraische Erweiterungen zur Verfügung stellt, war eine Untersuchung für Polynome über diesem Zahlentyp nicht möglich.

SINGULAR ist ein Computer Algebra System für Polynomberechnungen, welches den Schwerpunkt auf die Anforderungen der Kommutativen Algebra, algebraischen Geometrie und Singularitätentheorie gelegt hat. Entwickelt wurde es vom SINGULAR Team des Fachbereichs Mathematik der Universität Kaiserslautern. Für Polynome über den ganzen Zahlen benutzt SINGULAR den Algorithmus der NTL, für Polynome über einer algebraischen Erweiterung wird ein nicht modularer Algorithmus ausgeführt (siehe Abschnitt 5.4).

5.2 Verbesserung der Implementierung

Wie in Abschnitt 4.3.3 besprochen, wurde zur Verbesserung der Laufzeit ein *cache* für den erweiterten euklidischen Algorithmus und der Divides-Funktor anstelle der teuren Pseudo-Division eingeführt. Die Auswirkungen dieser Änderun-

gen auf die Laufzeit untersuchen wir mit Hilfe der Dateienfamilie mit aufsteigendem Grad des *ggT*. Die Ergebnisse sind in Abbildung 5.1 zu sehen.

Der linke Plot vergleicht den alten Algorithmus von Brown ohne Verbesserungen mit der Version, die nur den *cache* benutzt und unserer neuen Implementierung mit beiden Verbesserungen. Im rechten Plot sind die Laufzeiten der gleichen Versionen unseres Hybridansatzes für Polynome über algebraischen Erweiterungen zu sehen.

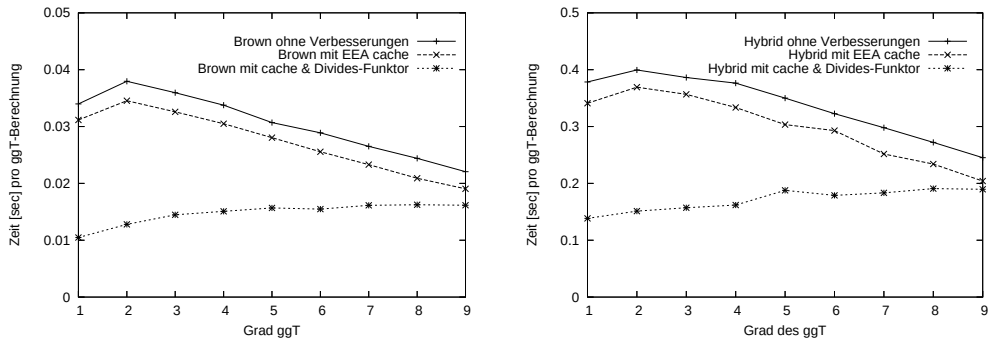


Abbildung 5.1: Polynome vom Grad 10; ganzzahlige Koeffizienten des *ggT* und der Kofaktoren 2000 Bit. Links drei Versionen des Algorithmus von Brown für Polynome über $\mathbb{Z}$. Rechts Versionen des Hybridansatzes für Polynome über einer algebraischen Erweiterung von $\mathbb{Z}$.

Wie wir sehen, haben die alten Versionen der beiden Algorithmen ab Grad 2 einen monoton fallenden Verlauf. Der Grund dafür ist, dass in der Verifikationsphase die Pseudo-Division benutzt wird. Ist die Differenz zwischen Polynom- und *ggT*-Grad sehr groß, werden die Polynome, nach Lemma 2.18, mit einem sehr großen Faktor erweitert, wodurch die Laufzeit der Testdivision schlechter wird.
Durch den *cache* im erweiterten euklidischen Algorithmus folgt eine Parallelverschiebung der beiden Kurven nach unten. Da wir den *ggT* von 50 verschiedenen Polynompaaren berechnen, werden während der ersten *ggT*-Berechnung, die Ergebnisse des EEA in den *cache* geschrieben. Da unglückliche Primzahlen sehr unwahrscheinlich sind, ändern sich die verwendeten Primzahlen im Allgemeinen nicht und für die weiteren Polynompaare wird der EEA nicht mehr ausgeführt. Stattdessen können die Zahlen aus dem *cache* gelesen werden. Somit kommt es zu einer Laufzeitverbesserung. Benutzen wir in der Verifikationsphase den Divides-Funktor, wird die teure Erweiterung während der Pseudo-Division vermieden. Da

die Zeitersparnis für einen kleinen *ggT* am größten ist, werden die Kurven nach unten verschoben und um das rechte Ende gedreht. Den neuen Verlauf der Kurven diskutieren wir ausführlich im Rahmen der Abbildung 5.5.

5.3 Polynome über $\mathbb{Z}$

Zuerst untersuchen wir die Laufzeiten der *ggT*-Algorithmen für Polynome über den ganzen Zahlen $\mathbb{Z}$. Wir werden unsere Version des Algorithmus von Brown, auf die wir unter Abschnitt 3.1.2 ausführlich eingegangen sind, mit den Laufzeiten der Algorithmen der NTL-Bibliothek und SINGULAR vergleichen. Um den großen Zeitunterschied zu verdeutlichen, zeigen einige Plots zusätzlich die Laufzeiten des nicht modularen Algorithmus aus NUMERIX. Als nicht modularer *ggT* wurde der Subresultanten-Algorithmus von G. Collins implementiert. Eine Beschreibung dieses Verfahrens liefert Henri Cohen in [7] Kapitel 3.3 (Algorithmus 3.3.1).

5.3.1 Untersuchungen für gleiche Bitlängen

In diesem Abschnitt befassen wir uns mit Polynompaaren deren Kofaktor- und *ggT*-Koeffizienten dieselbe Bitlänge aufweisen. Diese Situation ist überschaubarer, da wir nur die Kenngrößen Bitlänge, Polynomgrad und Grad des *ggT* verändern können.
Die linken Plots der Abbildungen zeigen den Vergleich unserer Implementierung des Algorithmus von Brown mit den Laufzeiten der Ansätze aus SINGULAR und der NTL-Bibliothek. Die rechten Plots stellen dem Algorithmus von Brown die Laufzeiten des nicht modularen Ansatzes aus NUMERIX gegenüber.

Wachsende Bitlänge

Zunächst untersuchen wir die Auswirkungen einer Veränderung der Bitlängen der Kofaktoren und des *ggT*. Dazu betrachten wir Polynompaare vom Grad 25 mit einem *ggT* vom Grad 1 und lassen die Bitlänge der Koeffizienten des *ggT* und der Kofaktoren ansteigen.
Im linken Plot der Abbildung 5.2 ist zu erkennen, dass mit steigender Bitlänge die Laufzeiten aller Algorithmen polynomiell zunehmen. Die experimentellen Ergeb-

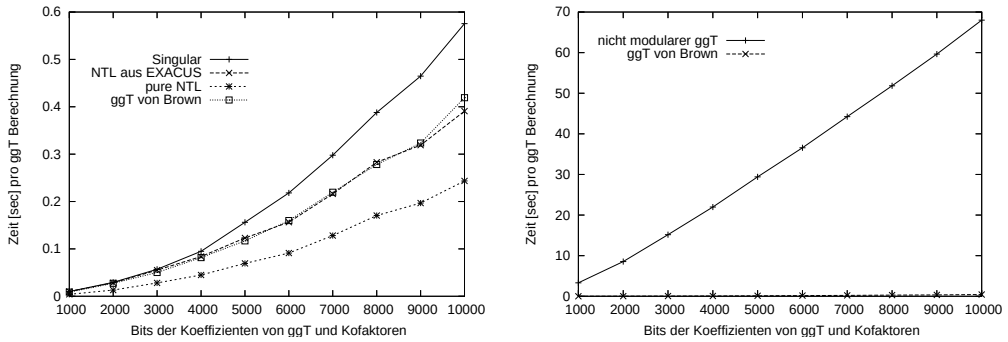


Abbildung 5.2: Laufzeiten der Algorithmen bei wachsender Bitlänge. Polynome vom Grad 25; Grad des *ggT* 1.

nisse des Algorithmus von Brown bestätigen die theoretische Komplexitätsanalyse aus 3.1.3, wonach die Laufzeit mit wachsender Bitlänge quadratisch ansteigt. Durch die größere Bitlänge benötigen die modularen Algorithmen mehr Primzahlen, um die Koeffizienten korrekt darzustellen. Diese Tatsache allein würde ein konstantes Wachstum erklären. Zusätzlich wird aber mit steigender Bitlänge jede einzelne modulare Berechnung, jede Anwendung des chinesischen Restsatzes und die Testdivision teurer, da mit größeren Koeffizienten gerechnet werden muss.
Man sieht weiterhin, dass sich die Laufzeit einer *ggT*-Berechnung unseres generischen Algorithmus nur um einen konstanten Faktor (kleiner zwei) vom spezialisierten Algorithmus der NTL unterscheidet. Berücksichtigen wir die zusätzlichen Kosten der Transformation in NTL Zahlentypen (NTL aus EXACUS), so sind beide Verfahren gleich gut. Wie bereits erwähnt, benutzt SINGULAR ebenfalls den Algorithmus der NTL. Offensichtlich sind dort die Kosten der Transformation teurer.
Der rechte Plot und die unterschiedliche Skalierung der Ordinaten verdeutlicht die Überlegenheit der modularen Algorithmen gegenüber der nicht modularen Berechnung.

Wachsender Polynomgrad

Analysieren wir nun, welchen Effekt eine Erhöhung des Polynomgrades hat. Um eine möglichst lange PRS zu erhalten, besitzen die Polynompaare einen *ggT* vom Grad 1. Die Größe der Koeffizienten des *ggT* und der Kofaktoren beträgt 5000 Bit.

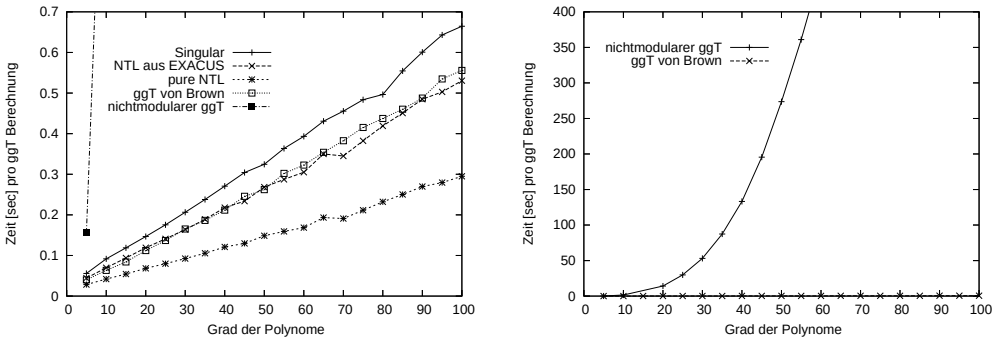


Abbildung 5.3: Laufzeiten der Algorithmen bei wachsendem Polynomgrad. Grad des *ggT* 1; Koeffizienten des *ggT* und der Kofaktoren 5000 Bit.

Der linke Plot der Abbildung 5.3 zeigt wiederum, dass sich die Laufzeiten der modularen Algorithmen nur um einen konstanten Faktor unterscheiden. Diese Differenzen hängen auch hier mit den unterschiedlichen Transformationskosten zusammen. Die modularen Algorithmen weisen, in diesem Untersuchungsbereich, ein lineares Wachstum der Laufzeit auf, da für einen größeren Polynomgrad mehr modulare Bilder berechnet werden müssen, bzw. der euklidische Algorithmus und die Testdivision mehr Schritte benötigen. Da die Bitlänge unverändert bleibt, sind die Kosten für jede einzelne Berechnung aber konstant. Asymptotisch werden wir aber nach Gleichung (3.4) ein quadratisches Wachstum erhalten, da die Laufzeit des euklidischen Algorithmus mit steigendem Polynomgrad quadratisch wächst. In unserem Untersuchungsbereich wird die Rechenzeit aber von den großen Koeffizienten dominiert, womit sich ein lineares Wachstum ergibt.
Der rechte Plot zeigt deutlich, wie sich der höhere Grad der Polynome exponentiell auf die Laufzeit des nichtmodularen Algorithmus auswirkt. Steigt der Grad der Polynome an, so verlängert sich bei konstantem *ggT*–Grad die PRS und das Koeffizientenwachstum vergrößert sich.

Wachsender Grad des *ggT*

Wir betrachten Polynompaare vom Grad 30 mit einer Koeffizientengröße von 1000 Bit und lassen den Grad des *ggT* von 1 auf 29 ansteigen.
Im linken Plot der Abbildung 5.4 fällt sofort die ungewöhnliche Laufzeitentwick-

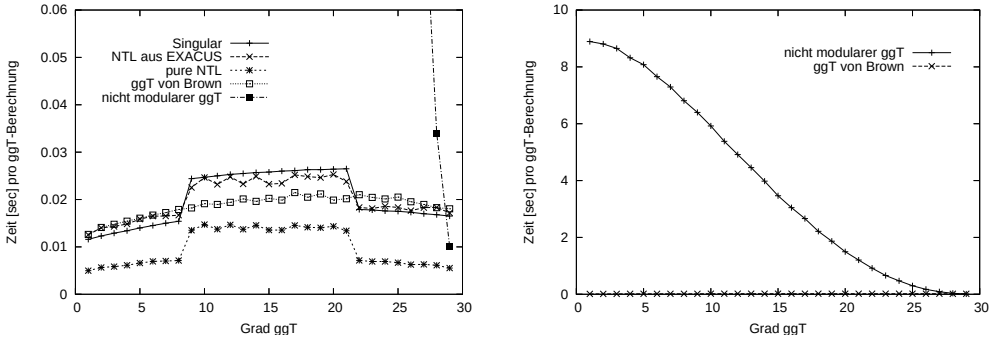


Abbildung 5.4: Laufzeiten der Algorithmen bei wachsendem Grad des *ggT*. Polynompaare vom Grad 30; Koeffizienten des *ggT* und der Kofaktoren 1000 Bit.

lung der NTL- und SINGULAR-Algorithmen auf. Für einen *ggT*-Grad zwischen 9 und 21 wird die Berechnung sprunghaft langsamer, wohingegen die Laufzeit unseres Algorithmus gleichmäßig verläuft. Die Vermutung liegt nahe, dass die NTL zwei verschiedene Algorithmen benutzt, abhängig vom Grad des *ggT*.

Der rechte Plot zeigt, dass die Laufzeit des nichtmodularen Ansatzes mit steigendem *ggT*-Grad sinkt. Durch einen größeren *ggT* verkürzt sich die PRS, wodurch das Koeffizientenwachstum eingeschränkt wird. Für einen *ggT*-Grad von 29 ist der nicht modulare Ansatz sogar dem *ggT* von Brown überlegen.

In Abbildung 5.5 ist zu sehen, wie sich die Gesamtzeit des Algorithmus von Brown zusammensetzt. Da sich der Polynomgrad und die Größe der Koeffizienten nicht ändern, bleibt die Berechnungszeit der modularen Bilder der Polynome konstant. Mit steigendem Grad des *ggT* benötigt der euklidische Algorithmus weniger Schritte, um diesen zu bestimmen. Demnach hat die Zeit des euklidischen Algorithmus einen fallenden Verlauf. Die Laufzeit des chinesischen Restsatzes ist genau entgegengesetzt, da er für einen größeren *ggT* mehr Koeffizienten rekonstruieren muss. Die Laufzeit der Testdivision läßt sich durch die Anzahl der Multiplikationen während einer Polynomdivision erklären. Ist der Grad des *ggT* ähnlich groß wie der der Kofaktoren, so treten im Laufe der Testdivision mehr Multiplikationen auf, als wenn der Grad des *ggT* sehr groß oder sehr klein ist (vgl. Komplexitätsanalyse 3.1.3).

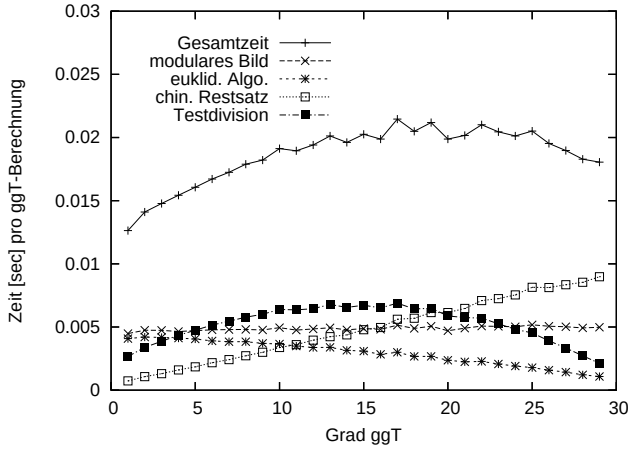


Abbildung 5.5: Einzelne Zeiten des modularen Algorithmus von Brown. Polynompaare vom Grad 30; Koeffizienten des *ggT* und der Kofaktoren 1000 Bit.

5.3.2 Untersuchungen für unterschiedliche Bitlängen

Gehen wir nun auf Polynompaare mit unterschiedlichen Koeffizientengrößen der Kofaktoren und des *ggT* ein.

Wachsende Bitlänge des *ggT*

Zuerst analysieren wir die Auswirkungen eines Anstiegs der Koeffizientengröße des *ggT*. Dazu betrachten wir Polynompaare vom Grad 30, deren Kofaktorkoeffizienten aus 500 Bit bestehen. Da die NTL-Bibliothek für einen mittleren *ggT*-Grad schlechtere Laufzeiten aufweist (vgl. Abbildung 5.4), betrachten wir eine erste Polynomreihe mit *ggT*-Grad 1 und eine zweite mit *ggT*-Grad 15. Die Ergebnisse sind im linken und rechten Plot der Abbildung 5.6 zu sehen.

Wie beide Plots zeigen, weisen alle Laufzeiten einen polynomiellen Anstieg auf. Auch diese beobachteten Laufzeiten bestätigen unsere theoretische Komplexitätsanalyse aus Abschnitt 3.1.3. Diese besagt, dass der Algorithmus von Brown mit wachsender Koeffizientengröße des *ggT* asymptotisch quadratisches Wachstum aufweist. Für unseren Algorithmus werden wir in Abbildung 5.7 eine genauere Erklärung dieser Laufzeit liefern. Weiterhin verdeutlichen die Ergebnisse, dass sich unser Algorithmus von Brown auch für diese Versuchsreihe nur um einen konstanten Faktor vom NTL-Algorithmus unterscheidet.

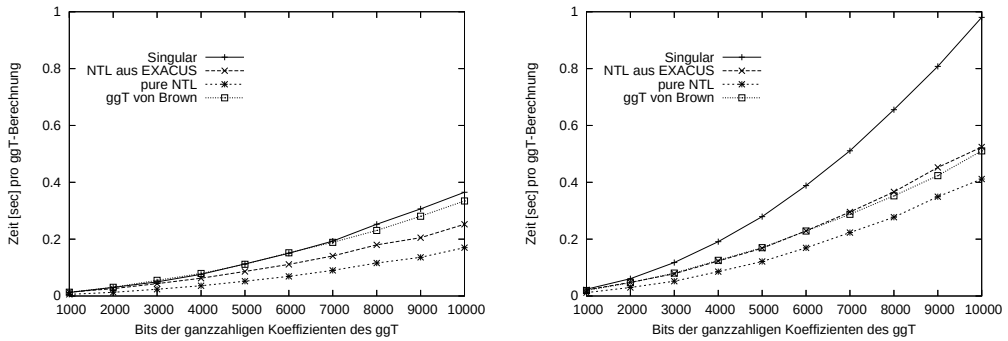


Abbildung 5.6: Laufzeiten der Algorithmen bei anwachsender Bitlänge des *ggT*. Polynome vom Grad 30, ganzzahlige Koeffizienten der Kofaktoren 500 Bit. Links Grad des *ggT* 1, rechts 15.

Ein Vergleich der Laufzeiten des linken (*ggT*-Grad 1) und rechten (*ggT*-Grad 15) Plots zeigt, dass die Berechnung des größeren *ggT* mehr Zeit benötigt. Zum einen ist das auf die längere Laufzeit der Testdivision zurückzuführen (vgl. (3.4)), zum anderen muss der chinesische Restsatz mehr Koeffizienten rekonstruieren, da der *ggT* von höherem Grad ist. Wiederum zeigen die Plots die Unterschiede des Algorithmus der NTL in Abhängigkeit vom Grad des *ggT*. Die Algorithmen der NTL und SINGULAR verlieren für die Berechnung des *ggT* vom Grad 15 mehr Zeit als der Algorithmus von Brown. War unser Algorithmus für einen *ggT*-Grad von 1 schlechter als der NTL-Algorithmus inklusive Transformationskosten, sind beide Laufzeiten für einen *ggT*-Grad von 15 auf demselben Niveau.

In Abbildung 5.7 sind die Zeiten der wichtigsten Komponenten des *ggT*-Algorithmus von Brown zu sehen. Hier für einen *ggT*-Grad von 15.
Wie wir sehen, wachsen die Laufzeiten der Berechnung des modularen Bildes und des chinesischen Restsatzes polynomiell an. Nach der Komplexitätsanalyse 3.1.3 ergibt sich asymptotisch quadratisches Wachstum. Aufgrund der steigenden Bitlänge benötigt der Algorithmus erstens mehr Primzahlen, um die Koeffizienten des *ggT* zu rekonstruieren und zweitens wird jede einzelne Berechnung des modularen Bildes und des chinesischen Restsatzes teurer. Insgesamt ergibt sich demnach für die Berechnung dieser Komponenten, wie auch für die Testdivision, ein polynomielles Wachstum. Die Laufzeit des euklidischen Algorithmus wächst hingegen linear, da er über dem modularen Zahlentyp arbeitet. Folglich ist die Ko-

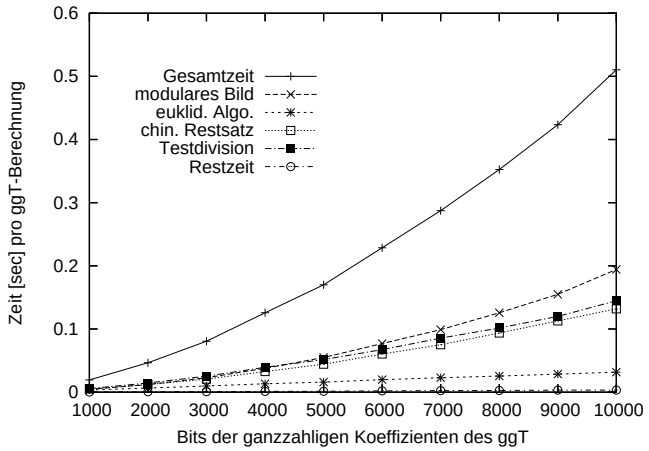


Abbildung 5.7: Einzelne Zeiten des modularen Algorithmus von Brown. Polynome vom Grad 30, Grad des *ggT* 15 und ganzzahlige Koeffizienten der Kofaktoren 500 Bit.

effizientengröße der Ausgangspolynome für die Laufzeit bedeutungslos. Da die übrige Zeit des Gesamtalgorithmus (Restzeit) vernachlässigbar klein ist, können wir davon ausgehen, dass alle relevanten Komponenten aufgeführt sind.

Wachsende Bitlänge der Kofaktoren

Läßt man die Bitlänge der Kofaktorkoeffizienten ansteigen, schneidet unser Algorithmus von Brown im Vergleich mit NTL und SINGULAR noch besser ab. Wieder wurden Polynompaare vom Grad 30 erzeugt. Jetzt bestehen die Koeffizienten des *ggT* aus 500 Bit. Für den *ggT*-Grad 1 ergeben sich die Laufzeiten des linken Plots der Abbildung 5.8, für den *ggT*-Grad 15 sind die Ergebnisse im rechten Plot abgetragen.

Für die Berechnung des *ggT* vom Grad 1 weisen alle Algorithmen ein lineares Wachstum der Laufzeit auf, wobei der pure Algorithmus der NTL nur minimal besser ist als unsere Implementierung. Durch die ansteigenden Transformationskosten der Zahlentypen weisen die Laufzeiten von SINGULAR und des NTL Algorithmus aus EXACUS eine größere Steigung auf. Berechnen wir den *ggT* vom Grad 15, ist der Algorithmus von Brown dem Ansatz der NTL sogar überlegen. Im Vergleich zum linken Plot haben sich die Zeiten des Brown Algorithmus nur leicht verschlechtert. Diese Beobachtung ist erneut auf die langsamere Testdivisi-

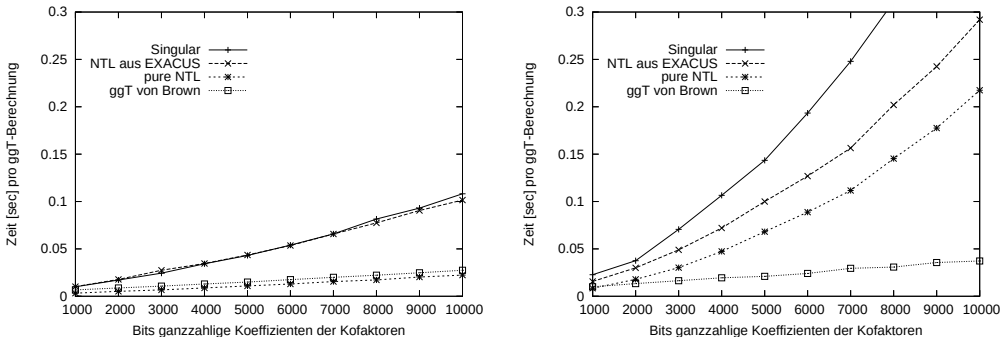


Abbildung 5.8: Laufzeiten der Algorithmen bei anwachsender Bitlänge der Kofaktoren. Polynome vom Grad 30, ganzzahlige Koeffizienten des *ggT* 500 Bit. Links Grad des *ggT* 1, rechts 15.

on und die häufigere Anwendung des chinesischen Restsatzes zurückzuführen. Im Vergleich dazu verlieren die Algorithmen der NTL und SINGULAR wieder mehr Zeit. Desweiteren entwickeln sich diese Laufzeiten nicht linear wie im linken Plot, sondern steigen nun fast quadratisch an. Auch hieraus kann man schließen, dass die NTL verschiedene Algorithmen zur *ggT*-Berechnung benutzt.

Für den *ggT* vom Grad 15 schauen wir uns wieder die einzelnen Zeiten des Algorithmus von Brown an. Aus Abbildung 5.8 wissen wir, dass die Laufzeiten von Brown, im Vergleich zum Algorithmus der NTL, sehr langsam ansteigen. In Ab-

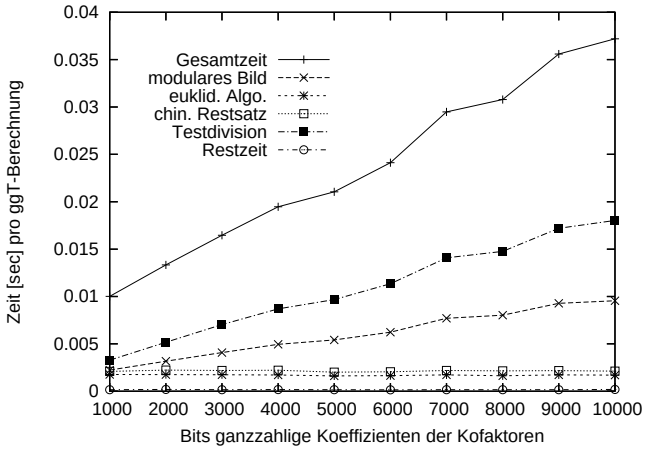


Abbildung 5.9: Einzelne Zeiten des modularen Algorithmus von Brown. Polynome vom Grad 30, Grad des *ggT* 15 und ganzzahlige Koeffizienten des *ggT* 500 Bit.

bildung 5.9 sind demnach sehr kleine Zeitdifferenzen abgetragen.
Da die Bitlänge des *ggT* konstant bleibt, benötigt der Algorithmus immer dieselbe Anzahl an Primzahlen. Demzufolge bleiben auch die Laufzeiten des euklidischen Algorithmus und des chinesischen Restsatzes konstant. Wie Abbildung 5.9 zeigt, hat nur die Berechnung des modularen Bildes und die Testdivision einen ansteigenden Verlauf. Der Grund dafür ist, dass diese beiden Komponenten mit den Ausgangspolynomen arbeiten, deren Koeffizientenlänge ansteigt. Der lineare Anstieg des Gesamtalgorithmus, sowie die Entwicklung der Einzellaufzeiten lassen sich wieder gut an unserer theoretischen Komplexitätsanalyse 3.1.3 ablesen.

Wachsender Grad des *ggT*

Wir untersuchen nun, wie sich eine Änderung des *ggT*-Grades auf die Laufzeit auswirkt, wenn die Kofaktorkoeffizienten (5000 Bit) größer sind als die des *ggT* (500 Bit). Wieder wurden zwei Polynomreihen untersucht. Erstens Polynompaare vom Grad 30 und zweitens vom Grad 50.

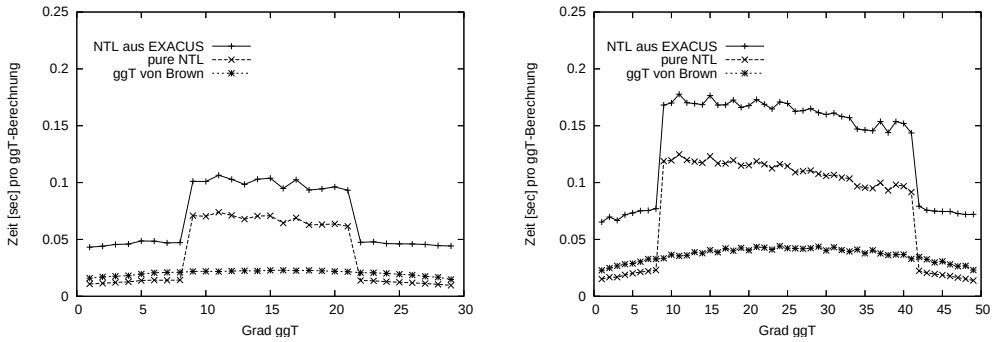


Abbildung 5.10: Laufzeiten der Algorithmen bei anwachsendem Grad des *ggT*. Ganzzahlige Koeffizienten des *ggT* 500, der Kofaktoren 5000. Links Polynompaare vom Grad 30, rechts vom Grad 50.

Wie Abbildung 5.10 zeigt, tritt auch für diese Analyse die sprunghafte Laufzeitverschlechterung der NTL auf. Für mittlere *ggT*-Grade ist unser Algorithmus von Brown sogar schneller als der pure NTL-Algorithmus. Für Ausgangspolynome vom Grad 50 verschärft sich diese Laufzeitverschlechterung noch weiter. Erstens ist der Laufzeitverlust einer einzelnen *ggT*-Berechnung größer und zweitens sind mehr *ggT*-Grade davon betroffen als im linken Plot. Die Ergebnisse lassen vermuten, dass der schnellere *ggT*-Algorithmus der NTL nur für die neun kleinsten bzw.

größten *ggT*-Grade benutzt wird.
Die Laufzeiten der einzelnen Komponenten des Algorithmus von Brown sind vergleichbar mit den Ergebnissen aus Abbildung 5.5.

5.4 Polynome über algebraischen Erweiterungen

Analysieren wir nun die Laufzeiten der *ggT*-Algorithmen für Polynome über einer algebraischen Erweiterung der ganzen Zahlen vom Grad 2. Wir werden die Laufzeiten unseres Hybridansatzes mit den modularen Algorithmen von Langemyr-McCallum und Encarnacion vergleichen. Alle drei Ansätze wurden unter Abschnitt 3.2 vorgestellt. Darüber hinaus haben wir die Laufzeiten der nichtmodularen Algorithmen von NUMERIX und SINGULAR ermittelt. Die NUMERIX-Bibliothek benutzt den euklidischen Algorithmus mit Hilfe der unter Lemma 2.18 eingeführten Pseudo-Division als nichtmodularen Algorithmus. Um unnötigen Rechenaufwand zu vermeiden, werden nach jeder Pseudo-Division des euklidischen Algorithmus überflüssige Skalare aus den Restpolynomen entfernt. SINGULAR benutzt dasselbe Verfahren, allerdings werden hier die überflüssigen Skalare nicht entfernt. Da sich diese Methode als sehr langsam herausstellte, verzichten wir an dieser Stelle auf eine Darstellung der Laufzeiten in den Plots.
Wenn wir in diesem Abschnitt von der Bitlänge der Koeffizienten sprechen, meinen wir die Länge jedes ganzzahligen Koeffizienten der algebraischen Erweiterung. Der Radikant besteht in diesen Untersuchungen immer aus derselben Bitlänge wie die ganzzahligen Koeffizienten des *ggT*.

5.4.1 Wachsende Bitlänge des *ggT*

Untersuchen wir zuerst die Entwicklung der Laufzeiten, wenn sich die Bitlängen der Koeffizienten des *ggT* ändern. Zu diesem Zweck betrachten wir Polynompaare bestehend aus Kofaktoren vom Grad 10 mit Koeffizienten aus 2000 Bit und einem *ggT* vom Grad 1 mit ansteigender Koeffizientenlänge (500 bis 5000 Bit). Die Ergebnisse sind in Abbildung 5.11 zu sehen.

Der linke Plot zeigt, dass die Laufzeiten des Algorithmus von Langemyr-McCallum (LM) ähnlich gut sind wie die des Hybridansatzes. Für eine größere Bitlänge des

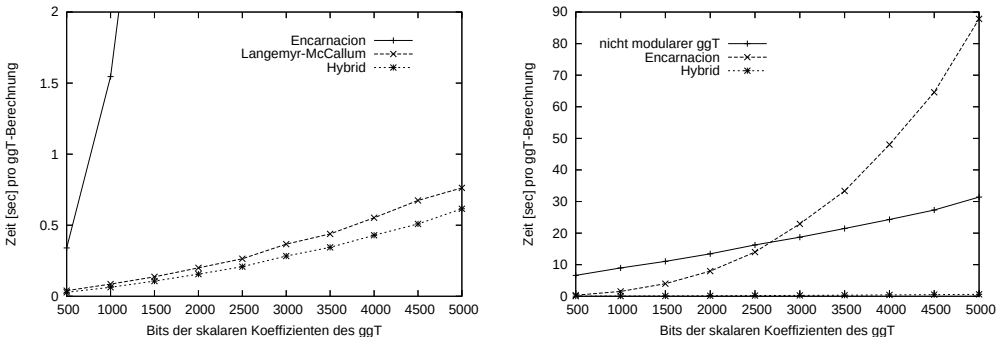


Abbildung 5.11: Laufzeiten der Algorithmen bei anwachsender Bitlänge des *ggT*. Polynome vom Grad 10, Grad des *ggT* 1 und ganzzahlige Koeffizienten der Kofaktoren 2000 Bit.

ggT leidet der LM Algorithmus allerdings unter der Multiplikation mit dem größer werdenden *Denominator for Algebraic Integers (dfai)*. Encarnacion ist in seiner Komplexitätsanalyse (vgl. Abschnitt 3.2.5) zu dem Ergebnis gekommen, dass die Rechenzeit des LM–Algorithmus asymptotisch quadratisch mit der Bitlänge des *ggT* wächst. Die Rechenzeit des Algorithmus von Encarnacion nimmt, nach der theoretischen Analyse, kubisch mit der Bitlänge des *ggT* zu. Erklären läßt sich das anhand der Komplexität der *Rational Reconstruction* von Wang. Nach Gleichung (3.17) ist sie $O(mn\log^3(\mathcal{G}))$, wobei $\mathcal{G}$ die Bitlänge des *ggT* beschreibt.
Wie wir im rechten Plot sehen, ist der Algorithmus von Encarnacion für eine genügend große Bitlänge sogar langsamer als der nicht modulare Ansatz aus NUMERIX. Der Algorithmus von SINGULAR leidet stark unter den überflüssigen Skalaren und ist nicht konkurrenzfähig. Für eine Größe der *ggT*-Koeffizienten von 500 Bit benötigt er bereits 128 Sekunden für eine Berechnung. Die Laufzeit steigert sich linear auf 783 Sekunden für eine Koeffizientenlänge von 5000 Bit.

Die Zeiten der wichtigsten Komponenten unseres Hybridansatzes sind in Abbildung 5.12 zu sehen. Die Gesamtlaufzeit wächst, wie die Laufzeit des LM–Algorithmus, asymptotisch quadratisch. Durch die größere Bitlänge der Koeffizienten des *ggT* benötigt der modulare Hybridansatz mehr Primzahlen, um die Zahlen korrekt darstellen zu können. Zusätzlich werden auch hier (analog zu Abb. 5.7) die einzelnen Berechnungen der modularen Bilder, des chinesischen Restsatzes und der Testdivision teurer. Die Zeit, um die Ausgangspolynome zu normalisieren und den *dfai* zu bestimmen (Normalisierung & dfai) ist ebenfalls

leicht ansteigend. Die Laufzeit des Algorithmus von Wang besitzt bei einer Koeffizientenlänge von 4500 Bit einen Sprung. Vermutlich wird ab hier die *Rational Reconstruction* häufiger aufgerufen, da die akkumulierten Zeiten des chinesischen Restsatzes die unter 3.2.6 besprochene Schranke erreicht haben.

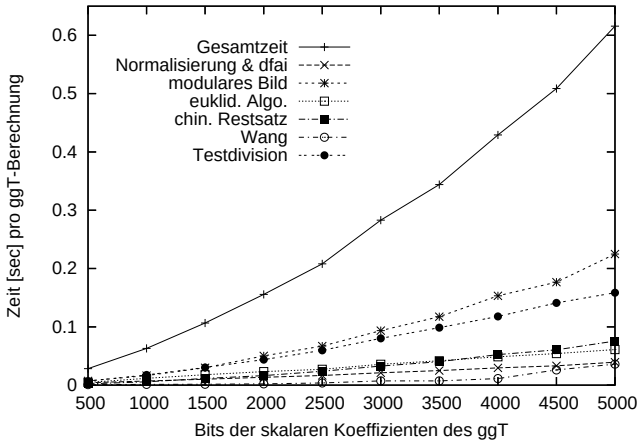


Abbildung 5.12: Einzelne Zeiten unseres Hybridansatzes. Polynome vom Grad 10, Grad des *ggT* 1 und ganzzahlige Koeffizienten der Kofaktoren 2000 Bit.

5.4.2 Wachsende Bitlänge der Kofaktoren

Lässt man dagegen die Bitlänge der Kofaktoren ansteigen, ergibt sich ein anderes Bild der Laufzeiten. Wir betrachten wieder Polynompaare vom Grad 10, die einen *ggT* vom Grad 1 besitzen. Diesmal beträgt die Länge der ganzzahligen Koeffizienten des *ggT* 1000 Bit und die der Kofaktoren variiert.

Wie der linke Plot der Abbildung 5.13 zeigt, verlaufen die Zeiten der drei modularen Algorithmen nahezu parallel und weisen nur eine leichte, lineare Steigung auf. Der LM–Algorithmus leidet für größere Koeffizienten wieder unter der Multiplikation mit dem *dfai* und verliert etwas Zeit im Vergleich zum Hybridansatz. Der Algorithmus von Encarnacion ist wieder der Langsamste, denn auch hier kostet die *Rational Reconstruction* von Wang viel Zeit. Da aber die Koeffizienten des *ggT* konstant bleiben und somit die Laufzeit von Wangs Algorithmus für alle Polynompaare gleich ist, steigt auch die Laufzeit von Encarnacions Algorithmus nur leicht an. Ein Vergleich mit Encarnacions Komplexitätsanalyse ist leider nicht

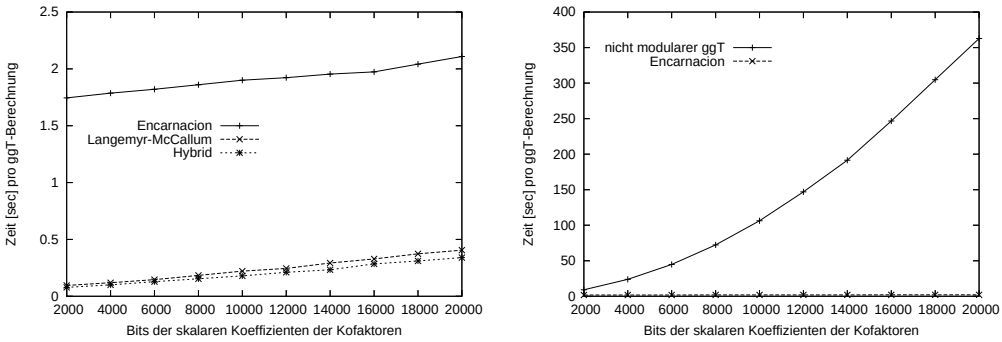


Abbildung 5.13: Laufzeiten der Algorithmen bei anwachsender Bitlänge der Kofaktoren. Polynome vom Grad 10, Grad des *ggT* 1 und ganzzahlige Koeffizienten des *ggT* 1000 Bit.

möglich, da er die Bitlänge der Ausgangspolynome nicht abgebildet hat.
Der rechte Plot verdeutlicht, dass das Koeffizientenwachstum für den nicht modularen Algorithmus eine größere Rolle spielt. Die Rechenzeit wächst mit zunehmender Bitlänge fast quadratisch an. Der Algorithmus von SINGULAR leidet noch stärker unter dem Koeffizientenwachstum. Für eine Länge von 2000 Bit benötigt er 240 Sekunden für eine *ggT*-Berechnung, für 14000 Bit sind es bereits über 100 Minuten (6538 Sekunden).

Betrachten wir in Abbildung 5.14 die Zeiten der einzelnen Komponenten unseres

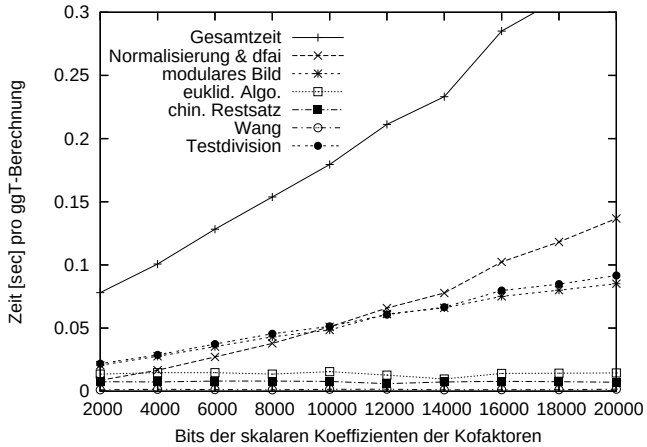


Abbildung 5.14: Einzelne Zeiten unseres Hybridansatzes. Polynome vom Grad 10, Grad des *ggT* 1 und ganzzahlige Koeffizienten des *ggT* 1000 Bit.

Hybridansatzes. Zu beachten ist, dass es sich wieder nur um sehr kleine Zeitdifferenzen handelt.
Die Normalisierung der Polynome und die Berechnung des *dfai* leiden am meisten unter dem Anstieg der Bitlänge. Für größere Koeffizienten wird die Multiplikation zur Normalisierung des Leitkoeffizienten natürlich wesentlich teurer. Ebenfalls einen ansteigenden Verlauf weisen die Berechnung der modularen Bilder und die Testdivision auf. Da die Länge der Kofaktorkoeffizienten für den euklidischen Algorithmus, den chinesischen Restsatz sowie Wangs Algorithmus keine Rolle spielen, sind die Laufzeiten dieser Komponenten konstant.

5.4.3 Wachsender Polynomgrad

Untersuchen wir nun die Laufzeitentwicklung bei steigendem Polynomgrad. Die Bitlänge der ganzzahligen Kofaktorkoeffizienten beträgt in dieser Untersuchung 500 Bit. Der *ggT* ist vom Grad 1 und besitzt Koeffizienten aus 2000 Bit.

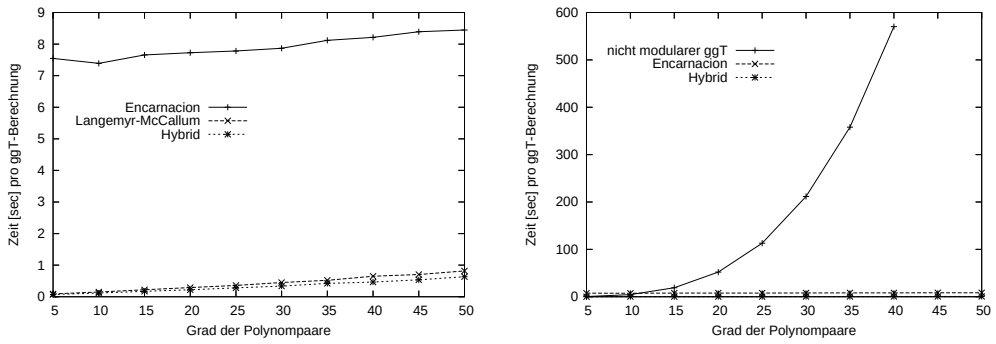


Abbildung 5.15: Laufzeiten der Algorithmen bei anwachsendem Polynomgrad. Die ganzzahligen Koeffizienten der Kofaktoren bestehen aus 500 Bit, die des *ggT* aus 2000 Bit. Der *ggT* ist vom Grad 1.

Die Laufzeiten der modularen Algorithmen in Abbildung 5.15 verlaufen wieder nahezu parallel und weisen eine leichte, lineare Steigung auf. Aus den bekannten Gründen ist Encarnacions Algorithmus der Langsamste und der Algorithmus von Langemyr-McCallum etwas schlechter als der Hybridansatz. Nach der Komplexitätsanalyse 3.2.5 sollte die Rechenzeit der Algorithmen quadratisch mit dem Polynomgrad ansteigen. Da unsere Polynome aber von verhältnismäßig

kleinem Grad sind, dominieren die großen Koeffizienten die Laufzeit. Der rechte Plot verdeutlicht, wie schnell sich die Rechenzeit des nichtmodularen Algorithmus bei wachsendem Polynomgrad verschlechtert. Durch einen höheren Polynomgrad verlängert sich die PRS, wodurch das Koeffizientenwachstum stark vergrößert wird.

Abbildung 5.16 zeigt, wie sich die Gesamtlaufzeit des Hybridansatzes ergibt. Da die Bitlängen der Koeffizienten und der Grad des *ggT* fest sind, bleiben die Laufzeiten des chinesischen Restsatzes, von Wangs Algorithmus, sowie der Berechnung des *dfai* konstant. Die Laufzeit der Berechnung der modularen Bilder und der Testdivision steigt hingegen linear an. Der Grund dafür ist, dass für höhergradige Polynome mehr modulare Bilder bestimmt werden müssen und die Testdivision mehr Schritte benötigt. Asymptotisch ist die Laufzeit des euklidischen Algorithmus quadratisch im Polynomgrad, für unseren Untersuchungsbereich dominieren allerdings die großen Koeffizienten die Rechenzeit. Auch die Normalisierung der Polynome verzeichnet einen leichten Anstieg, da mehr Koeffizienten mit dem Normalisierungsfaktor multipliziert werden.

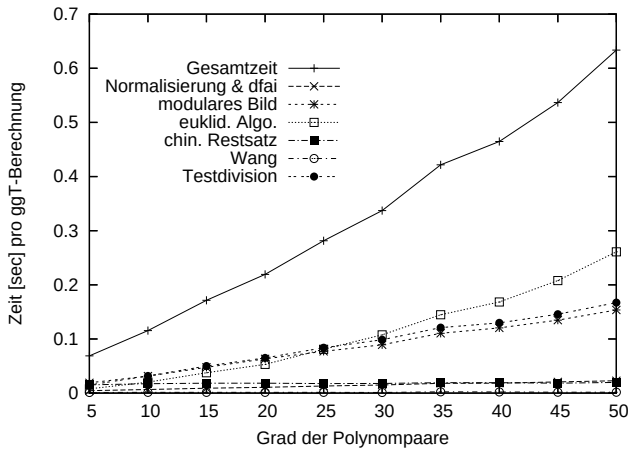


Abbildung 5.16: Einzelne Zeiten unseres Hybridansatzes. Ganzzahlige Koeffizienten der Kofaktoren 500 Bit, des *ggT* 2000 Bit; Grad des *ggT* 1.

Kapitel 6

Zusammenfassung

Wir haben vier verschiedene modulare Algorithmen zur *ggT*-Berechnung für univariate Polynome entwickelt. Alle Methoden erfüllen die drei wichtigsten Ziele der EXACUS-Bibliothek:

- **Exaktheit:**
  Die Algorithmen berechnen immer das mathematisch richtige Ergebnis.
- **Vollständigkeit:**
  Sie berechnen für jede sinnvolle Eingabe, d.h. einschließlich der degenerierten Fälle, das richtige Ergebnis.
- **Effizienz:**
  Die Algorithmen sind effizient gemessen an ihren Laufzeiten. Sie sind z.B. den alten nichtmodularen Methoden aus EXACUS weit überlegen (vgl. Kapitel 5).

Das Kapitel Benchmarks zeigt weiterhin, dass die Laufzeiten unseres modularen Algorithmus für Polynome über den ganzen Zahlen mit den Laufzeiten der renommierten Zahlentheorie Bibliothek NTL konkurrieren können. Für einige Fälle erzielen wir sogar bessere Ergebnisse.

Für Polynome über algebraischen Erweiterungen war unseres Wissens bislang kein modularer *ggT*-Algorithmus frei Verfügbar. Wir haben die theoretischen Ansätze von Langemyr–McCallum [22] und Encarnacion [11] generisch implemen-

tiert und daraus einen eigenen Hybridansatz entwickelt. Wie sich in Kapitel 5 herausstellte, ist unser Hybridansatz den Algorithmen von Langemyr–McCallum und Encarnacion für alle Untersuchungen überlegen. Die experimentellen Laufzeiten stimmen außerdem weitgehend mit den theoretischen Komplexitätsanalysen überein.

Dennoch gibt es Möglichkeiten zur Weiterentwicklung der geleisteten Arbeit.

Multivariater *ggT*:
Wünschenswert wäre die Implementierung eines modularen *ggT*-Algorithmus für multivariate Polynome. Für ganzzahlige Polynome kann man das Vorgehen folgendermaßen beschreiben (vgl. Brown [6]):
Wir gehen davon aus, dass wir zwei Polynome $F_1, F_2 \in (\mathbb{Z}_p[y])[x]$ gegeben haben, wobei p eine Primzahl ist. Wir betrachten F_1 und F_2 also als Polynome mit Koeffizienten aus $\mathbb{Z}_p[y]$. Diese univariaten Koeffizienten lassen sich modulo Ideale der Form $\langle y-b\rangle$ mit $b \in \mathbb{Z}_p$ so reduzieren, dass wir Elemente des endlichen Körpers $\mathbb{Z}_p$ erhalten. Die ursprünglich bivariaten Polynome sind nun Elemente des euklidischen Ringes $\mathbb{Z}_p[x]$, in dem wir den *ggT* wieder mit Hilfe des euklidischen Algorithmus berechnen können. Die Reduktion eines univariaten Koeffizientenpolynoms $k(y)$ modulo $\langle y-b\rangle$ ist dabei einfach die Auswertung $k(b)$ von k an der Stelle b. Diese Rechnung wiederholen wir wieder für eine bestimmte Anzahl von glücklichen Idealen $\langle y-b\rangle$. Um aus den berechneten homomorphen Bildern wieder auf den tatsächlichen *ggT* zurückschließen zu können, verwenden wir eine polynomiale Version des Chinesischen Restsatzes. Durch rekursive Anwendung dieses Vorgehens kann so auch ein *ggT* in mehr als zwei Variablen berechnet werden.

Erweiterungen von höherem Grad:
Da die präsentierten Algorithmen generisch geschrieben sind, sollten sie auch auf Polynome über höhergradigen Erweiterungen anwendbar sein. Ausführlich getestet wurden sie allerdings nur für die – für unsere Anwendung relevanten – Erweiterungen von Grad 2. Ein detaillierter Test für Erweiterungen von höherem Grad sollte also noch entwickelt werden.

Das Nullteilerproblem in R_p:
Wie in Abschnitt 4.3.2 erwähnt, ist unser Ansatz, das Problem mit Hilfe des Gradvektors der PRS zu lösen, eine erste naive Herangehensweise. Eine Weiterentwicklung dieser Lösung sollte direkt an der Funktion `is_zero` ansetzen. Diese testet, ob der Leitkoeffizient des nächsten Polynoms der PRS Null bzw. ein Nullteiler ist. Sie sollte so verändert werden, dass zusätzlich zwischen diesen beiden Fällen unterschieden wird. Ist der modulare Leitkoeffizient eines Polynoms in der PRS ein Nullteiler, können wir die Berechnungen abbrechen und diese Primzahl als unglücklich verwerfen. Wie dieser Nullteilertest allgemeingültig zu implementieren ist und wie man die Information an den *ggT*-Algorithmus zurückliefert, wäre Gegenstand weiterführender Überlegungen.

Literaturverzeichnis

[1] Mathew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[2] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. Exacus: Efficient and exact algorithms for curves and surfaces. In *Proc. of the 13th Annual European Symposium on Algorithms(ESA 2005)*, pages 155–166. Springer, 2005.

[3] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In *ESA 2002, Lecture Notes in Computer Science*, pages 174–186, 2002.

[4] E. Berberich, M. Hemmer, L. Kettner, E. Schömer, and N. Wolpert. An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In *In Proc. 21th Annu. Sympos. Comput. Geom.*, pages 99–106, 2005.

[5] S. Bosch. *Algebra*. Springer, Berlin, 1996.

[6] W. S. Brown. On euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM*, 18:478–504, 1971.

[7] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993.

[8] G. E. Collins and M. J. Encarnacion. Efficient rational number reconstruction. *J. Symbolic Computation 20*, pages 287–297, 1995.

[9] A. Eigenwillig, M. Kerber, and N. Wolpert. Fast and exact geometric analysis of real algebraic plane curves. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC 2007)*, pages 151–158, 2007.

[10] A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert. Complete, exact, and efficient computations with cubic curves. In *In Proc. 20th Annu. Sympos. Comput. Geom.*, pages 409–418, 2004.

[11] M. J. Encarnacion. Computing gcds of polynomials over algebraic number fields. *J. Symbolic Computation*, 20:299–313, 1995.

[12] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[13] M. Hemmer and D. Hülse. Traits classes for polynomial gcd computation over algebraic extensions. Technical Report ACS-TR-241405-03, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, Germany, 2007.

[14] Michael Hemmer. Algebraic foundations. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.3 edition, 2007.

[15] Michael Hemmer. *Exact Computation of the Adjaceny Graph of an Arrangement of Quadrics*. Ph.d. dissertation, Johannes Gutenberg-Universität Mainz, 2007.

[16] T. W. Hungerford. *Algebra*, volume 5. Springer-Verlag, 2003.

[17] M. Jazayeri, R. Loos, and D.R. Musser. *Generic Programming: International Seminar on Generic Programming Dagstuhl Castle*. Springer, Germany, 2000.

[18] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *In Proc. 15th Annu. Sympos. Comput. Geom.*, pages 351–359, 1999.

[19] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1981.

[20] G. Lamé. Note sur la limite du nombre des divisions daans la recherche du plus grand commun diviseur entre deux nombres entiers. *Comptes Rendus de l'Académie des Sciences Paris*, 19:867–870, 1844.

[21] S. Lang. *Algebra*, volume 4. Springer-Verlag, 2005.

[22] L. Langemyr and S. McCallum. The computation of polynomial gcd's over an algebraic number field. *J. Symbolic Computation*, 8:429–448, 1989.

[23] K. Mehlhorn and S. Näher. *LEDA: A Plattform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[24] M. E. Pohst. *Computational algebraic Number Theory*. DMV Seminar Band 21. Springer-Verlag, 1993.

[25] V. Shoup. *A computational introduction to number theory and algebra*. Cambridge Univ. Press, 2005.

[26] P. S. Wang. A p-adic algorithm for univariate partial fractions. *Proceedings of SYMSAC '81, ACM Press*, pages 212–217, 1981.

[27] P. S. Wang, M.J.T. Guy, and J.H. Davenport. P-adic reconstruction of rational numbers. *SIGSAM Bulletin*, pages 2–3, 1982.

[28] P. J. Weinberger and L. P. Rothschild. Factoring polynomials over algebraic number fields. *ACM Transactions on Mathematical Software*, 2.