Faktorisieren von Polynomen über $\mathbb{F}_q$: ein Vergleich neuerer Verfahren

Diplomarbeit im Fach Informatik

bearbeitet von

Jürgen Gerhard
geb. am 27. April 1967 in Kulmbach

Beginn der Arbeit: 1. Februar 1994
Abgabe der Arbeit: 15. Juli 1994

angefertigt am

Lehrstuhl für Automatentheorie und formale Sprachen (Informatik I)
Institut für Mathematische Maschinen und Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:

Dr. Helmut Meyn
Priv.-Doz. Dr. Volker Strehl

Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 15. Juli 1994 ....

Inhaltsverzeichnis

Einleitung

Die Computer-Algebra ist ein immer mehr an Bedeutung und Anwendungsfeldern gewinnendes Fachgebiet im Grenzbereich zwischen theoretischer Informatik und Mathematik. Leistungsfähige Programmsysteme wie Maple, Axiom oder Mathematica unterstützen Anwenderinnen und Anwender aus Mathematik, Informatik, Physik, Chemie, Biologie und den Ingenieurwissenschaften bei der Manipulation von Formeln und Gleichungssystemen.

Eine speziell beim Lösen von polynomialen Gleichungen (oder Gleichungssystemen) immer wiederkehrende Aufgabe für ein Computer-Algebra-System ist das *Faktorisieren von Polynomen.* Ist beispielsweise die Gleichung $f(x, y) = 0$ für $x, y \in \mathbb{Q}$ zu lösen, wobei f ein Polynom in zwei Variablen mit ganzzahligen Koeffizienten ist, welches sich als Produkt $f = f_1 f_2$ mit $\deg f_1, \deg f_2 > 0$ schreiben läßt, dann ist die Lösungsmenge der Gleichung $f(x, y) = 0$ die Vereinigung aller Lösungen der beiden Gleichungen $f_1(x, y) = 0$ und $f_2(x, y) = 0$. Damit ist das Problem, ein polynomiale Gleichung vom Grad $n := \deg f$ zu lösen, auf die Lösung zweier polynomialer Gleichungen *kleineren Grades* als n zurückgeführt.

Der erste Schritt beim Lösen polynomialer Gleichungen ist daher meist die Faktorisierung des betreffenden Polynoms in irreduzible Faktoren:

$$f = {g_1}^{e_1} \cdot \ldots \cdot {g_t}^{e_t}$$

Dabei ist f ein Polynom in einer oder mehreren Unbestimmten mit Koeffizienten aus $\mathbb{Z}$ oder einem Körper, etwa $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, einer algebraischen Erweiterung von $\mathbb{Q}$ oder einem endlichen Körper $\mathbb{F}_q$. Das Faktorisierungsproblem läßt sich für viele der genannten Fälle (außer für $\mathbb{R}$ und $\mathbb{C}$) auf ein spezielles Faktorisierungsproblem zurückführen: die Faktorisierung von Polynomen in einer Variablen über einem endlichen Körper $\mathbb{F}_q$.

Seit Ende der 60er Jahre gibt es für dieses Problem einen deterministischen Algorithmus nach E.R. Berlekamp, der ein Polynom $f \in \mathbb{F}_q[X]$ vom Grad n mit $\mathcal{O}(n^3 q)$ Körperoperationen in $\mathbb{F}_q$ faktorisiert. Eine probabilistische Variante kommt mit erwarteten $\mathcal{O}(n^3 + n^2 \log n \cdot \log q)$ Körperoperationen aus. Beide Verfahren beruhen auf einer *Linearisierung* des Faktorisierungsproblems: zunächst muß eine Basis des Lösungsraums eines linearen Gleichungssystems bestimmt werden, aus welcher anschließend durch ggT-Berechnungen die Faktorisierung von f erhalten werden kann. H. Niederreiter stellte 1992 ein Verfahren vor, welches viele Ähnlichkeiten mit Berlekamps Algorithmus aufweist und ebenfalls die Lösung eines linearen Gleichungssystems erfordert. Seine Laufzeit liegt – zumindest für kleine Charakteristik – in derselben Größenordnung wie die von Berlekamps Verfahren.

Ein großer Nachteil bei beiden Algorithmen ist, daß relativ kostenaufwendig eine $n \times n$-Matrix eliminiert werden muß. Das klassische Gauß-Verfahren benötigt dafür $\mathcal{O}(n^3)$ Körperoperationen in $\mathbb{F}_q$. Es sind zwar asymptotisch schnellere Verfahren bekannt, die sich aber erst für sehr große n in der Praxis rentieren. Ein probabilistischer Ansatz, der die Elimination umgeht, geht auf D. Wiedemann (1986) zurück. Mit erwarteten $\mathcal{O}(n \cdot \mathrm{BB}(n))$ Körperoperationen kann die Lösung $x \in {\mathbb{F}_q}^n$ eines nichtsingulären, linearen Gleichungsssytems $Ax = b$ für $A \in {\mathbb{F}_q}^{n\times n}$, $b \in {\mathbb{F}_q}^n$ berechnet werden, wobei $\mathrm{BB}(n)$ die Kosten für eine Anwendung der Koeffizientenmatrix A auf einen beliebigen Vektor in ${\mathbb{F}_q}^n$ bezeichnen.

E. Kaltofen et al. (1991, 1994) haben diese Methode an den bei Berlekamp und Niederreiter interessanten Fall der Lösung eines singulären Gleichungssystems $Ax = 0$ angepaßt. Mit derselben asymptotischen Laufzeit wie bei Wiedemann wird eine zufällige Lösung x berechnet. Eine Anwendung von A auf einen Vektor kann im Berlekamp- wie im Niederreiter-Fall auf Polynomarithmetik reduziert werden, und für kleine Charakteristik kann in beiden Fällen $\mathrm{BB}(n) = M(n)$, die Kosten für eine Multiplikation zweier Polynome über $\mathbb{F}_q$ vom Grad n, gewählt werden. Bei Verwendung schneller Multiplikationsalgorithmen (z.B. Karatsuba oder FFT) führt dieser Ansatz zu einem asymptotisch schnelleren Faktorisierungsverfahren als die Gauß-Elimination.

Das Ziel der Arbeit war neben der Beschreibung der theoretischen Hintergründe eine Implementation der verschiedenen Algorithmen und eine Messung der Laufzeiten.

Im ersten Kapitel werden die theoretischen Grundlagen für die Verfahren von Berlekamp und Niederreiter sowie die Algorithmen selbst in ihrer klassischen Form vorgestellt und ein Vergleich gezogen. Das zweite Kapitel beschreibt den probabilistischen Algorithmus von Wiedemann zur Lösung linearer Gleichungssysteme, die darauf beruhende Idee von Kaltofen & Lobo (1994) und ihre Anwendung auf die Faktorisierungsalgorithmen von Berlekamp und Niederreiter.

Die in den ersten beiden Kapiteln beschriebenen Verfahren wurden für endliche Körper der Charakteristik 2 in der Programmiersprache C++ implementiert. Im dritten Kapitel ist diese Implementation dargestellt. Es enthält auch eine Bedienungsanleitung für das Programm und dokumentiert einige Testläufe.

Kapitel 1

Faktorisierungsalgorithmen

1.1 Problemstellung und Übersicht

Sei $\mathbb{F}_q$ ein endlicher Körper der Charakteristik p mit $q = p^r$ Elementen. Der Polynomring $\mathbb{F}_q[X]$ in einer Variablen über $\mathbb{F}_q$ ist ein Hauptidealring und damit faktoriell. Jedes Polynom $f \in \mathbb{F}_q[X]$ läßt sich (bis auf die Reihenfolge) eindeutig in ein Produkt von irreduziblen Faktoren zerlegen:

$$f = c \cdot \prod_{1 \leq i \leq t} g_i^{e_i} \tag{1.1}$$

Dabei sind $c \in \mathbb{F}_q$, $t \in \mathbb{N}$, $e_i \in \mathbb{N} \setminus \{0\}$ $(1 \leq i \leq t)$ und die $g_i \in \mathbb{F}_q[X]$ normiert, irreduzibel und paarweise verschieden sind. Zu lösen ist das folgende

Faktorisierungsproblem: *Zu gegebenem f sind die Größen c, t, g_i und e_i $(1 \leq i \leq t)$ in der Darstellung (1.1) zu berechnen.*

Im allgemeinen wird das Problem noch weiter unterteilt (der Einfachheit halber nehmen wir an, daß f normiert, also $c = 1$ ist):

- **quadratfreie Zerlegung** (engl. *squarefree decomposition*): Zerlegung von f in Potenzen von quadratfreien Polynomen:

  $$f = \prod_{1 \leq i \leq \deg f} f_i^i \tag{1.2}$$

  wobei $f_i \in \mathbb{F}_q[X]$ normiert und quadratfrei ist, d.h. es gibt kein $a \in \mathbb{F}_q[X]$ mit $\deg a \geq 1$ und $a^2 | f$. Den *quadratfreien Anteil von f* notieren wir mit $f_{\boxtimes} := \prod_{1 \leq i \leq \deg f} f_i$.

- **„distinct degree"-Faktorisierung**: Zerlegung eines quadratfreien Polynoms f in Faktoren, deren Primfaktoren alle denselben Grad haben:

$$f = \prod_{1 \le i \le \deg f} f_i \tag{1.3}$$

so daß $f_i \in \mathbb{F}_q[X]$ ein Produkt normierter, irreduzibler Polynome vom Grad i ist.

- **„equal degree"-Faktorisierung**: Zerlegung der Endprodukte f_i der „distinct degree"-Faktorisierung in irreduzible Faktoren.

- **„splitting"**: Zerlegung eines Polynoms $f \in \mathbb{F}_q[X]$ in zwei nichttriviale Faktoren: $f = f_1 \cdot f_2$ mit $\deg f_1 > 0$ und $\deg f_2 > 0$. Die vollständige Faktorisierung (1.1) kann daraus durch rekursive Anwendung oder durch Generierung mehrerer solcher „splits" und anschließender *Verfeinerung* (engl. *refinement*) der (partiellen) Faktorisierung durch sukzessive ggT-Bildung gewonnen werden.

Bei den derzeit existierenden Faktorisierungsverfahren treten unterschiedliche Kombinationen der oben genannten Schritte auf. Die weitaus meisten Algorithmen gehen jedoch davon aus, daß das zu faktorisierende Polynom f bereits quadratfrei ist. Vor deren Anwendung ist die Zerlegung (1.2) durchzuführen; das eigentliche Verfahren wird dann auf die quadratfreien Faktoren f_i angewandt.

Für die quadratfreie Zerlegung eines Polynoms f über Körpern der Charakteristik 0 gibt es mehrere Verfahren, die darauf beruhen, den größten gemeinsamen Teiler von f mit seiner formalen Ableitung f' zu berechnen. Das effizienteste davon ist der Algorithmus von Yun, dessen Laufzeit im westenlichen durch die ggT-Berechnung bestimmt wird (vgl. [GCL92], [vzGS92]). Für $\deg f = n$ beträgt sie $\mathcal{O}(n^2)$ Körperoperationen bei Verwendung der Standard-Polynomarithmetik bzw. $\mathcal{O}(M(n) \log n)$ bei Verwendung schneller Multiplikationstechniken (Karatsuba oder FFT, vgl. [AHU75]). Dabei bezeichnet $M(n)$ die für die Multiplikation zweier Polynome vom Grad n benötigte Anzahl von Körperoperationen.

Der Yun-Algorithmus erfordert für die Verwendung über endlichen Körpern kleine Änderungen, da die Ableitung f' eines nichtkonstanten $\mathbb{F}_{p^r}$-Polynoms f verschwindet, wenn f ein Polynom in X^p ist. In diesem Fall muß ein Polynom f_1 mit ${f_1}^{p^e} = f$ für ein $e \ge 1$ und ${f_1}' \ne 0$ gefunden und das Verfahren auf f_1 angewandt werden. Die Laufzeit wird dadurch jedoch nicht wesentlich beeinflußt (vgl. dazu auch die Diskussion in [Mig92]).

Die meisten derzeitigen Faktorisierungsverfahren basieren auf Berechnungen im Faktorring $R := \mathbb{F}_q[X]/(f)$, genauer: in einer speziellen Unter-Algebra dieses

Rings, der sogenannten *Berlekamp-Algebra*. Vor einer weiteren Klassifizierung der Faktorisierungsalgorithmen wird daher zunächst auf diese Algebra eingegangen.

1.2 Die Berlekamp-Algebra und Berlekamps Algorithmus

Sei $f \in \mathbb{F}_q[X]$ ein normiertes, quadratfreies Polynom vom Grad $\deg f = n > 0$ und $f = g_1 \cdot \ldots \cdot g_t$ seine Primfaktorzerlegung in paarweise verschiedene, irreduzible, normierte Polynome $g_i \in \mathbb{F}_q[X]$. Sei weiterhin $d_i := \deg g_i$ $(1 \leq i \leq t)$. Der Faktorring $R := \mathbb{F}_q[X]/(f)$ ist ein n-dimensionaler $\mathbb{F}_q$-Vektorraum mit kanonischer Basis $\{1 + (f), x + (f), \ldots, x^{n-1} + (f)\}$, also eine $\mathbb{F}_q$-Algebra der Dimension n. Wir identifizieren die Elemente von R mit ihren kanonischen Repräsentanten $\in \mathbb{F}_q[X]$ vom Grad kleiner als n und die Ringoperationen entsprechend mit koeffizientenweiser Addition und Multiplikation modulo f.

Der chinesische Restesatz liefert eine andere, für unsere Zwecke interessantere Darstellung von R:

$$R = \mathbb{F}_q[X]/(f) \cong \mathbb{F}_q[X]/(g_1) \times \ldots \times \mathbb{F}_q[X]/(g_t) \tag{1.4}$$

Der zugehörige Isomorphismus ist

$$\Phi : \begin{cases} R & \longrightarrow \quad \mathbb{F}_q[X]/(g_1) \times \ldots \times \mathbb{F}_q[X]/(g_t) \\ h & \mapsto \quad (h \bmod g_1, \ldots, h \bmod g_t) \end{cases} \tag{1.5}$$

Wegen der Irreduzibilität der g_i sind die Faktorringe $\mathbb{F}_q[X]/(g_i)$ Körper vom Erweiterungsgrad d_i über $\mathbb{F}_q$. Jedem Element h von R läßt sich nach (1.5) eineindeutig ein t-Tupel von Polynomen $(a_1, \ldots, a_t)$ mit $\deg a_i < d_i$ zuordnen, nämlich $a_i = h \bmod g_i$ $(1 \leq i \leq t)$. Den Skalaren (konstanten Polynomen) $a \in \mathbb{F}_q \subseteq R$ entsprechen dabei die t-Tupel $(a, \ldots, a)$. Addition, Multiplikation und F_q-Skalarmultiplikation in R entsprechen *komponentenweiser* Addition, Multiplikation bzw. Skalarmultiplikation von k-Tupeln:

$$\begin{aligned} (a_1, \ldots, a_t) + (b_1, \ldots, b_t) &= (a_1 + b_1, \ldots, a_t + b_t) \\ (a_1, \ldots, a_t) \cdot (b_1, \ldots, b_t) &= (a_1 \cdot b_1 \bmod g_1, \ldots, a_t \cdot b_t \bmod g_t) \\ \lambda \cdot (a_1, \ldots, a_t) &= (\lambda \cdot a_1, \ldots, \lambda \cdot a_t) \quad (\lambda \in \mathbb{F}_q) \end{aligned}$$

Im folgenden wird nicht mehr zwischen den beiden isomorphen Darstellungen von Elementen von R als $\mathbb{F}_q$-Polynome vom Grad $< n$ einerseits und als t-Tupel von Polynomen andererseits unterschieden.

Die Einheiten des Rings R sind genau diejenigen t-Tupel $(a_1, \ldots, a_t)$, die in *jeder* Komponente invertierbar sind, d.h. $a_i \neq 0$ $(1 \leq i \leq t)$. Um das Polynom f in zwei nichttriviale Faktoren zu zerlegen, genügt es, ein in R (modulo f) nicht invertierbares Element (Polynom) $h \neq 0$ zu finden. Denn es gilt:

$$a_i = h \bmod g_i = 0 \Leftrightarrow g_i | h \quad (1 \leq i \leq t)$$

und daher

$$\mathrm{ggT}(f, h) = \prod_{g_i | h} g_i = \prod_{1 \leq i \leq t;\ a_i = 0} g_i \tag{1.6}$$

für $h = (a_1, \ldots, a_t)$. Wegen $h \neq 0$ ist mindestens ein a_i ungleich 0 und damit der ggT nicht trivial.

Die nicht invertierbaren Elemente von R sind jedoch relativ dünn gestreut. Analog zur Berechnung der Euler'schen φ-Funktion bei den natürlichen Zahlen ergibt sich für die Anzahl der Nichteinheiten von R die Formel

$$q^n \cdot (1 - \prod_{1 \leq i \leq t} (1 - \frac{1}{q^{d_i}}))$$

(vgl. [Wie86]). Deshalb gehen wir zu einer Teilmenge von R über, in der es leichter ist, eine Nichteinheit zu finden.

Definition 1.1 (Berlekamp-Algebra) *Die Menge $\mathcal{B}$ aller t-Tupel $(a_1, \ldots, a_t)$ mit $a_i \in \mathbb{F}_q$ $(1 \leq i \leq t)$ (konstante Polynome!) heißt die* (relative*) Berlekamp-Algebra *von f (über $\mathbb{F}_q$).*

Das folgende Lemma charakterisiert die Berlekamp-Algebra genauer zeigt, daß es sich wirklich um eine Algebra handelt.

Lemma 1.2

i. $\mathcal{B}$ ist eine Unter-Algebra von R der Dimension t über $\mathbb{F}_q$

ii. $\mathcal{B} = \{h \in R;\ h^q = h\}$

*vgl. Bemerkung 1.7

Beweis:

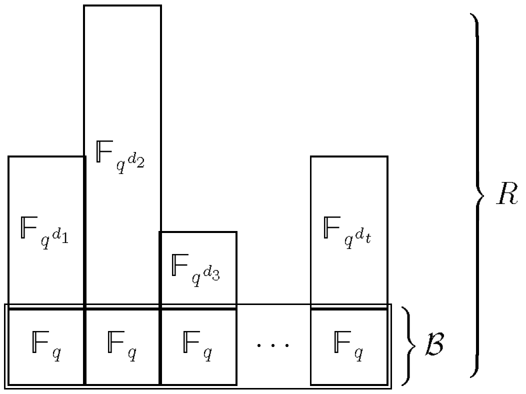
i. Klar ist: $\mathbb{F}_q \subseteq \mathcal{B}$, denn $a = (a, \ldots, a) \in \mathcal{B}$ für $a \in \mathbb{F}_q$, d.h. $\mathbb{F}_q$ kann als die „Diagonale" der Berlekamp-Algebra aufgefaßt werden. Da alle Algebra-Operationen komponentenweise erfolgen, ist $\mathcal{B}$ unter $+$, $\cdot$ und $\mathbb{F}_q$-Skalarmultiplikation abgeschlossen. Als $\mathbb{F}_q$-Algebra ist $\mathcal{B}$ demnach isomorph zu $\underbrace{\mathbb{F}_q \times \ldots \times \mathbb{F}_q}_{t \text{ mal}}$.

ii. Sei $h = (a_1, \ldots, a_t) \in R$.

$$\begin{aligned} h^q = h \quad &\Leftrightarrow \quad (a_1, \ldots, a_t)^q = (a_1, \ldots, a_t) \\ &\Leftrightarrow \quad (a_1^q, \ldots, a_t^q) = (a_1, \ldots, a_t) \\ &\Leftrightarrow \quad a_1, \ldots, a_t \in \mathbb{F}_q \\ &\Leftrightarrow \quad h \in \mathcal{B} \end{aligned}$$

□

Die Berlekamp-Algebra $\mathcal{B}$ als Teilmenge von R läßt sich bildlich veranschaulichen:



Jeder vertikale Balken stellt dabei eine Komponente $\mathbb{F}_q[X]/(g_i) \cong \mathbb{F}_{q^{d_i}}$ des direkten Produkts (1.4) dar, die Höhe des Balkens den Erweiterungsgrad d_i über $\mathbb{F}_q$ (im Beispiel also $d_1 = 3$, $d_2 = 5$, $d_3 = 2$ und $d_t = 3$). Die Algebra $\mathcal{B}$, der „Sockel" der Algebra R, ist das t-fache direkte Produkt des Grundkörpers $\mathbb{F}_q \subseteq \mathbb{F}_{q^{d_i}}$ der einzelnen Komponenten.

Die Bedeutung der Berlekamp-Algebra für die Faktorisierung ergibt sich aus folgender Tatsache:

Lemma 1.3 *Für $h \in \mathcal{B}$ gilt*

$$f = \prod_{a \in \mathbb{F}_q} ggT(h - a, f) \tag{1.7}$$

Beweis: Sind $h = (a_1, \ldots, a_t) \in \mathcal{B}$ und $a \in \mathbb{F}_q$, so ist $h - a = (a_1 - a, \ldots, a_t - a)$. Nach (1.6) gilt $\operatorname{ggT}(h-a, f) = \prod_{a_i = a} g_i$. Jedes g_i kommt folglich in dem Produkt auf der rechten Seite von (1.7) in genau einem Faktor vor (nämlich in dem mit $a_i = a$). Also gilt $\prod_{a \in \mathbb{F}_q} \operatorname{ggT}(h-a, f) = \prod_{1 \le i \le t} g_i = f$. □

Die Faktorisierung (1.7) liefert genau dann eine nichttriviale Zerlegung von f, wenn $h \neq (a, \ldots, a) = a$ für ein $a \in \mathbb{F}_q$ ist, d.h. h ist nicht konstant. Für kleine Körpergrößen q läßt sie sich in „vernünftiger“ Zeit berechnen.

Wie aber findet man ein nichtkonstantes $h \in \mathcal{B}$? Bezeichne $\varphi : h \mapsto h^q$ den *Frobenius-Endomorphismus* von R. Nach Lemma 1.2 ist $\mathcal{B} = \{h \in R;\ h^q = h\}$, der Kern des F_q-Algebra-Endomorphismus

$$\beta := \varphi - \mathrm{id} : \begin{cases} R & \longrightarrow \quad R \\ h & \mapsto \quad h^q - h \end{cases}$$

Dieser ist insbesondere $\mathbb{F}_q$-linear, hat also eine Darstellung als $n \times n$-Matrix $B = Q - I$ bzgl. der Standard-Polynombasis $1, x, \ldots, x^{n-1}$, wobei Q die dem Frobenius-Endomorphismus φ zugeordnete Matrix und I die $n \times n$-Einheitsmatrix ist. Die Matrix Q heißt auch die zu f gehörige *Berlekamp-Matrix* über $\mathbb{F}_q$. Nach der Dimensionsformel ist

$$\operatorname{rang}_{\mathbb{F}_q} B = n - \dim_{\mathbb{F}_q} \ker B = n - \dim_{\mathbb{F}_q} \mathcal{B} = n - t$$

Insbesondere ist f genau dann irreduzibel, wenn $\operatorname{rang}_{\mathbb{F}_q} B = n - 1$ ist.

Der klassische Berlekamp-Algorithmus berechnet mit bekannten Methoden der linearen Algebra (Gauß-Elimination) eine Basis $\{h_1, \ldots, h_t\}$ von $\ker B = \mathcal{B}$. Falls $t > 1$ ist, werden danach so lange Zerlegungen wie in (1.7) mit $h = h_1, h_2, \ldots$ durchgeführt, bis alle t Faktoren von f gefunden sind.

Algorithmus 1.4 (Berlekamp)

{ *input:* $f \in \mathbb{F}_q[X]$ *normiert, quadratfrei,* $\deg f = n$ }

{ *output:* $g_1, \ldots, g_t$, *die irreduziblen Faktoren von* f }

I. (setup) *Berechne die Koeffizienten* q_{ij} *von*

$$X^{qj} \equiv \sum_{0 \le i < n} q_{ij} X^i \bmod f \quad (0 \le j < n)$$

in der Polynombasisdarstellung von R, *und setze* $B := Q - I$, *wobei* $Q = (q_{ij})_{0 \le i,j < n}$†.

†In der Literatur wird meist die transponierte Matrix $Q^{\mathbf{T}}$ verwendet, für einen Vergleich mit dem Niederreiter-Verfahren ist die Matrix Q in dieser Form jedoch besser geeignet.

II. *Berechne mittels Gauß-Elimination den Rang $n-t$ von B und eine Basis $\{h_1, \ldots, h_t\}$ von $\ker B = \{h \in R; Bh = 0\}$ (o.E. sei $h_1 = 1$, vgl. Lemma 1.2).*

III. *Setze factorbase $:= \{f\}$.*

IV. *Wiederhole die Schritte V. und VI. für alle $a \in \mathbb{F}_q$ und $(1 < k \leq t)$ so lange, bis #factorbase $= t$:*

V. *Setze oldfactorbase := factorbase und factorbase $:= \emptyset$.*

VI. (split) *Für alle $b \in$ oldfactorbase berechne $g := ggT(h_k - a, b)$. Falls $g \neq 1, b$, setze factorbase := factorbase $\cup \{g, \frac{b}{g}\}$, andernfalls factorbase := factorbase $\cup \{b\}$.*

VII. *output factorbase.*

Für die Korrektheit von Algorithmus 1.4 bleibt zu zeigen, daß das Abbruchereignis „#factorbase $= t$“ in Schritt IV. tatsächlich eintritt. Dafür benötigen wir noch eine Definition:

Definition 1.5 (separierende Menge) *Eine Teilmenge $S \subseteq \mathcal{B}$ heißt* separierende Menge *für f, falls für jedes Paar i, j $(1 \leq i < j \leq t)$ ein $s \in S$ mit $s \bmod g_i \neq s \bmod g_j$ existiert.*

Wird in den letzten Schritten III. - VII. des Berlekamp-Algorithmus anstelle der Basis $\{h_1, \ldots, h_t\}$ eine beliebige separierende Menge S verwendet, dann ist damit sichergestellt, daß die Abbruchbedingung schließlich zutrifft und das Ergebnis in Schritt VII. tatsächlich die vollständige Faktorisierung von f ist.

Jede Basis $h_1, \ldots, h_t$ der Berlekamp-Algebra $\mathcal{B}$ *ist* jedoch separierend: Sei $b_1, \ldots, b_t$ die kanonische Basis von $\mathcal{B}$, die das Kongruenzensystem

$$b_k \equiv 1 \ (\mathrm{mod} g_k), \quad b_l \equiv 0 \ (\mathrm{mod} g_l) \ (l \neq k) \tag{1.8}$$

löst, d.h. $b_k = (0, \ldots, 0, \underbrace{1}_{k}, 0, \ldots, 0)$ $(1 \leq k \leq t)$. Die Basis $\{h_1, \ldots, h_t\}$ hat bzgl. der kanonischen Basis die Darstellung

$$h_k = (h_{1k}, \ldots, h_{tk}) = \sum_{1 \leq i \leq t} h_{ik} b_i$$

wobei $h_{ik} = h_k \bmod g_i$. Wäre nun $h_k \bmod g_i = h_k \bmod g_j$ (d.h. $h_{ik} = h_{jk}$) für ein Paar $i < j$ und alle $1 \leq k \leq t$, dann ließe sich jedes Basiselement h_k als Linearkombination der $t-1$ Elemente $b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_{j-1}, b_{j+1}, \ldots, b_t, (b_i+b_j)$ darstellen - im Widerspruch zur linearen Unabhängigkeit der h_k.

Für die Laufzeit von Algorithmus 1.4 lassen sich folgende Schranken angeben:

- Die Kosten für die Aufstellung der Berlekamp-Matrix Q in Schritt I. betragen $\mathcal{O}((\log q + n)M(n))$ Operationen in $\mathbb{F}_q$ für die Berechnung von $X^q \bmod f, X^{2q} \bmod f, \ldots, X^{(n-1)q} \bmod f$. Dabei sind $M(n)$ die Kosten für eine Multiplikation zweier Polynome vom Grad $< n$ modulo f (d.h. eine Multiplikation in R).
- Die Elimination der $n \times n$-Matrix B benötigt $\mathcal{O}(n^3)$ Körperoperationen in $\mathbb{F}_q$.
- Der Zerlegungsschritt VI. wird maximal $q \cdot t$ mal durchgeführt. Ein solcher Schritt benötigt im wesentlichen so viele Körperoperationen wie eine ggT-Berechnung zweier $\mathbb{F}_q$-Polynome vom Grad $\leq n$, d.s. $\mathcal{O}(n^2)$ bei klassischer Polynomarithmetik bzw. $\mathcal{O}(M(n)\log n)$ bei Verwendung schnellerer Techniken.

Insgesamt ergibt sich eine Zeit-Komplexität von $\mathcal{O}(n^3+qtn^2) = \mathcal{O}(qn^3)$ Operationen mit klassischer Arithmetik bzw. $\mathcal{O}(n^3+qtM(n)\log n) = \mathcal{O}(n^3+qnM(n)\log n)$ Operationen mit schneller Multiplikation (unter Beachtung von $t = \mathcal{O}(n)$). Die Laufzeit des Berlekamp-Algorithmus ist also nicht polynomial, und bis heute ist auch kein deterministischer Algorithmus mit polynomialer Laufzeit zur Lösung des Faktorisierungsproblems bekannt.

Wegen des Faktors q in der Laufzeitanalyse ist der Berlekamp-Algorithmus in seiner klassischen, deterministischen Form nur für kleine Körpergrößen q geeignet. Einige Faktorisierungsalgorithmen verwenden aus diesem Grund andere separierende Mengen anstelle einer Basis von $\mathcal{B}$, die schneller zu einer kompletten Faktorisierung führen, z.B. der Algorithmus von Camion und ein Verfahren nach Shoup (vgl. [MBG$^+$93] §2.4). Andere Abwandlungen des Berlekamp-Verfahrens randomisieren die Berechnung der kompletten Faktorisierung aus einer Basis der Berlekamp-Algebra und erzielen damit sowohl in der Praxis als auch theoretisch wesentlich bessere (erwartete) Laufzeiten. Eine davon wird nun vorgestellt.

Sie geht zurück auf Berlekamp und beruht auf der folgenden Tatsache. Für einen endlichen Körper $\mathbb{F}_q$ *ungerader* Charakteristik (für Charakteristik 2 existiert ein Analogon, vgl. [GCL92]) zerfällt die multiplikative Gruppe ${\mathbb{F}_q}^* = \mathbb{F}_q \backslash \{0\}$ in zwei gleichgroße Teilmengen: Die Menge $U := \{a \in {\mathbb{F}_q}^*;\ \exists b \in {\mathbb{F}_q}^* : b^2 = a\}$ der

Quadrate und ihr relatives Komplement $N := \mathbb{F}_q{}^* \backslash U$, die Menge der *Nicht-Quadrate*. U ist eine Untergruppe von $\mathbb{F}_q{}^*$ vom Index 2 und der Kern des Gruppen-Endomorphismus $\alpha : \mathbb{F}_q{}^* \longrightarrow \mathbb{F}_q{}^*;\ a \mapsto a^{\frac{q-1}{2}}$, dessen Bild $\alpha(\mathbb{F}_q{}^*)$ nur aus den beiden Elementen $1, -1$ besteht (vgl. z.B. [IR82] Kap. 5 §1).

Die Idee des „large prime" Berlekamp-Algorithmus besteht nun darin, den Homomorphismus α auf die multiplikative Gruppe $\mathcal{B}^*$ der Berlekamp-Algebra, die t-fache direkte Potenz von $\mathbb{F}_q{}^*$ (vgl. Beweis zu Lemma 1.2), auszudehnen:

$$\alpha : \begin{cases} \mathcal{B}^* & \longrightarrow \quad \mathcal{B}^* \\ h = (a_1, \ldots, a_t) & \mapsto \quad h^{\frac{q-1}{2}} = (a_1{}^{\frac{q-1}{2}}, \ldots, a_t{}^{\frac{q-1}{2}}) \end{cases}$$

Dann gilt: $\alpha(\mathcal{B}^*) = \{1, -1\}^t$, und die Wahrscheinlichkeit, daß für ein zufällig und uniform gewähltes Element $h \in \mathcal{B}^*$ der Funktionswert $\alpha(h) = (1, \ldots, 1) = 1$ oder $\alpha(h) = (-1, \ldots, -1) = -1$, also $\mathrm{ggT}(\alpha(h) - 1, f)$ trivial ist, sinkt mit wachsendem q gemäß

$$\mathrm{Prob}(\alpha(h) \in \{1, -1\}) = \frac{\#\{1, -1\}}{\#\{1, -1\}^t} = 2^{(1-t)}$$

Algorithmus 1.6 („large prime" Berlekamp)

{ *input, output und Schritte I. und II. wie in Algorithmus 1.4* }

III. *Setze factorbase* $:= \{f\}$.

IV. *Wiederhole die Schritte V. bis VII. so lange, bis #factorbase* $= t$:

V. *Wähle zufällig* $\lambda_1, \ldots, \lambda_t \in \mathbb{F}_q$ *und setze* $h := \sum_{1 \le i \le t} \lambda_i h_i$.

VI. *Setze oldfactorbase := factorbase und factorbase* $:= \emptyset$.

VII. (split) *Für alle* $b \in$ *oldfactorbase berechne*
$f_1 := ggT(h, b)$, $f_2 := ggT(h^{\frac{q-1}{2}} - 1 \bmod f, b)$, $f_3 := \frac{b}{f_1 f_2}$
und setze factorbase := factorbase $\cup (\{f_1, f_2, f_3\} \backslash \{1\})$.

VIII. *output factorbase.*

Die Kosten eines Zerlegungsschritts VII. ergeben sich aus den $\mathcal{O}(\log q \cdot M(n))$ Körperoperationen für die Berechnung von $h^{\frac{q-1}{2}} \bmod f$ und den Kosten für die ggT's. Die erwartete Anzahl von Iterationen des Algorithmus ist höchstens so groß wie die Anzahl c unabhängig und zufällig gewählter Elemente

$h_1, \dots, h_c \in \mathcal{B}^{*\ddagger}$, die im Mittel benötigt werden, damit $\{\alpha(h_1), \dots, \alpha(h_c)\}$ eine separierende Menge ist. Sie beträgt nach [vzGS92] $\mathcal{O}(\log t)$, und damit hat Algorithmus 1.6 eine erwartete Laufzeit von

$$\mathcal{O}(n^3 + (\log q + 2)n^2 \log t) = \mathcal{O}(n^3 + \log q \cdot n^2 \log n)$$

Operationen im klassischen Fall bzw.

$$\mathcal{O}(n^3 + (\log q + 2\log n)M(n) \log t) = \mathcal{O}(n^3 + (\log q + \log n)M(n) \log n)$$

Operationen mit schneller Polynomarithmetik.

Bemerkung 1.7 Bei den Algorithmen 1.4 und 1.6 spielt der q-Frobenius-Endomorphismus $\varphi : h \mapsto h^q$ der $\mathbb{F}_q$-Algebra R eine zentrale Rolle. Wegen $\mathbb{F}_p \subseteq \mathbb{F}_q$ ist R auch eine $\mathbb{F}_p$-Algebra (der Dimension $r \cdot n$, wenn $q = p^r$). Die meisten Beobachtungen und Tatsachen dieses Abschnitts treffen direkt oder leicht modifiziert auf R als $\mathbb{F}_p$-Algebra zu, wenn an die Stelle des q-Frobenius-Morphismus der p-Frobenius-Morphismus $\overline{\varphi} : h \mapsto h^p$ und $\mathbb{F}_p$ an die Stelle von $\mathbb{F}_q$ tritt.
Analog zu Def. 1.1 wird die *absolute Berlekamp-Algebra* $\overline{\mathcal{B}}$ definiert als die Menge aller t-Tupel $(a_1, \dots, a_t)$ mit $a_i \in \mathbb{F}_p$ $(1 \le i \le t)$. $\overline{\mathcal{B}}$ ist eine $\mathbb{F}_p$-Unteralgebra von $\mathcal{B}$ (und damit von R) der Dimension t und besteht genau aus denjenigen Elementen h von R mit $h^p = h$. (1.7) gilt entsprechend, wenn $\mathbb{F}_q$ durch $\mathbb{F}_p$ ersetzt wird. Der Berlekamp-Algorithmus muß dahingehend modifiziert werden, daß die zu dem $\mathbb{F}_p$-linearen Operator $\overline{\beta} := \overline{\varphi} - \mathrm{id}$ gehörige $rn \times rn$-Matrix $\overline{B}$ (deren Rang $rn - t$ beträgt) aufgestellt und eliminiert wird. Hierbei sind die Koeffizienten des $\mathbb{F}_q$-Polynoms h ihrerseits wieder als polynomiale Ausdruücke über $\mathbb{F}_p$ zu betrachten, d.h. $\overline{B}$ ist die Darstellung von $\overline{\beta}$ bzgl. der Basis

$$\{1, \alpha, \dots, \alpha^{r-1}, X, \alpha X, \dots, \alpha^{r-1}X, \dots, X^{n-1}, \alpha X^{n-1}, \dots, \alpha^{r-1}X^{n-1}\} \qquad (1.9)$$

von R über $\mathbb{F}_p$, wobei α ein erzeugendes Element der Körpererweiterung $\mathbb{F}_q/\mathbb{F}_p$ ist. Für die „splits“ brauchen nur noch alle $a \in \mathbb{F}_p$ (statt $\in \mathbb{F}_q$) berücksichtigt zu werden. Analog ist Algorithmus 1.6 zu modifizieren.
Der große Nachteil der modifizierten Verfahren liegt darin, daß eine wesentlich größere Matrix eliminiert werden muß. Die Elimination der $n \times n$-Matrix benötigt $\mathcal{O}(n^3 A(r))$ $\mathbb{F}_p$-Operationen und Speicherplatz für $\mathcal{O}(n^2 r)$ Elemente von $\mathbb{F}_p$ gegenüber $\mathcal{O}(n^3 r^3)$ Operationen und Speicherbedarf $\mathcal{O}(n^2 r^2)$ für die Elimination der $nr \times nr$-Matrix. Hierbei bezeichnet $A(r)$ die Kosten für eine arithmetische Operation in $\mathbb{F}_q$, gemessen in $\mathbb{F}_p$-Operationen, also $A(r) = r^2$ bei Verwendung der Standard-Polynomarithmetik bzw. $A(r) = M(r) \log r$ mit schnelleren Techniken. Dafür wird die Berechnung der vollständigen Faktorisierung aus einer Basis der

‡Die Wahrscheinlichkeit dafür, daß $h \in \mathcal{B} \backslash \mathcal{B}^*$ in Schritt V. (und damit in $f_1 \neq 1$ Schritt VII.) ist, kann bei großem q vernachlässigt werden.

Berlekamp-Algebra erleichtert. Der Faktor q (bzw. $\log q = r \log p$) in der Laufzeitabschätzung für diesen letzten Teil der Algorithmen 1.4 und 1.6 (gemessen in $\mathbb{F}_q$-Operationen) verringert sich auf p (bzw. $\log p$).
Für die Aufstellung der Matrix $\overline{B}$ sind die Elemente $(\alpha^i X^j)^p = \alpha^{ip} X^{jp} \in R$ für $0 \leq i < r$ und $0 \leq j < n$ zu berechnen. Dies erfordert

- $\mathcal{O}(\log p + r)$ Operationen in $\mathbb{F}_q$ für die Berechnung von $\alpha^p, \alpha^{2p}, \ldots, \alpha^{(r-1)p}$ in $\mathbb{F}_q$,
- $\mathcal{O}((\log p + n)M(n))$ Operationen in $\mathbb{F}_q$ für die Berechnung von $X^p \bmod f, X^{2p} \bmod f, \ldots, X^{(n-1)p} \bmod f$, und
- $\mathcal{O}(n^2 r)$ Multiplikationen in $\mathbb{F}_q$ für die Berechnung der nr Produkte $\alpha^{ip} \cdot (X^{jp} \bmod f)$.

Zusammen sind das $\mathcal{O}((\log p + n)M(n) + n^2 r)$ Operationen in $\mathbb{F}_q$ oder $\mathcal{O}(((\log p + n)M(n) + n^2 r) \cdot A(r))$ Operationen in $\mathbb{F}_p$.
Damit beträgt die Laufzeit der „absoluten" Varianten der Algorithmen 1.4 bzw. 1.6

$$\mathcal{O}(n^3 r^3 + ptM(n) \log n \cdot A(r)) = \mathcal{O}(n^3 r^3 + p \cdot nM(n) \log n \cdot A(r))$$

bzw.

$$\mathcal{O}(n^3 r^3 + (\log p + \log n) M(n) \log t \cdot A(r)) = \mathcal{O}(n^3 r^3 + (\log p + \log n) M(n) \log n \cdot A(r))$$

Operationen in $\mathbb{F}_p$ mit schneller Polynomarithmetik.

Bemerkung 1.8 Ist das zu faktorisierende Polynom f nicht quadratfrei, dann liefert der Berlekamp-Algorithmus (und auch Algorithmus 1.6) nicht die vollständige Faktorisierung von f, sondern die Zerlegung in seine *Primärkomponenten*:

$$f = \prod_{1 \leq i \leq t} \overline{g_i}\ ; \quad \overline{g_i} = {g_i}^{e_i}$$

In diesem Fall ist nämlich $R \cong \mathbb{F}_q[X]/(\overline{g_1}) \times \ldots \times \mathbb{F}_q[X]/(\overline{g_t})$ nach dem Chinesischen Restesatz, wobei die einzelnen Komponenten $\mathbb{F}_q[X]/(\overline{g_i})$ nicht notwendigerweise Körper sind. Für $h = (a_1, \ldots, a_t) \in R$ ($a_i = h \mod \overline{g_i}$) gilt aber nach wie vor

$$\begin{aligned} h^q = h \quad &\Leftrightarrow \quad ({a_1}^q, \ldots, {a_t}^q) = (a_1, \ldots, a_t) \\ &\Leftrightarrow \quad a_i \in \mathbb{F}_q \text{ „}\subseteq\text{“ } \mathbb{F}_q[X]/(g_i) \text{ „}\subseteq\text{“ } \mathbb{F}_q[X]/(\overline{g_i}) \end{aligned}$$

(denn ${a_i}^q \equiv a_i \pmod{{g_i}^{e_i}} \Rightarrow {a_i}^q \equiv a_i \pmod{g_i}$). Also ist

$$\mathcal{B} := \{(a_1, \ldots, a_t) \in R;\ a_i \in \mathbb{F}_q\} = \{h \in R;\ h^q = h\}$$

und Berlekamps Algorithmus berechnet eine Basis von $\mathcal{B}$.
(1.7) gilt ebenfalls, und wenn ein irreduzibler Faktor g_i einen der ggT's auf der rechten Seite von (1.7) teilt, dann wird dieser ggT wegen

$$h \bmod g_i = a \text{ und } h \bmod {g_i}^{e_i} \in \mathbb{F}_q \Rightarrow h \bmod {g_i}^{e_i} = a$$

auch schon von $\overline{g_i} = {g_i}^{e_i}$ geteilt. Ein „split“ im Berlekamp-Algorithmus vermag demnach lediglich die Primärkomponenten $\overline{g_i}$ voneinander zu trennen.

1.3 Niederreiters Algorithmus

Der Berlekamp-Algorithmus und seine Varianten beruhen auf einer *Linearisierung* des Faktorisierungsproblems. Sie führen das per se nichtlineare Problem der Zerlegung eines Polynoms über dem endlichen Körper $\mathbb{F}_q$ in seine Primfaktoren zurück auf ein Problem der linearen Algebra über dem Grundkörper $\mathbb{F}_q$. Lange Zeit verwendeten fast alle Verfahren zu diesem Zweck den q-Frobenius-Endomorphismus der Algebra $R = \mathbb{F}_q[X]/(f)$. Im Jahre 1992 veröffentlichte Niederreiter einen anderen Ansatz zur Linearisierung des Faktorisierungsproblems, der statt der Gleichung $h^q = h$ in der Algebra R eine Differentialgleichung im Funktionenkörper $\mathbb{F}_q(X)$ betrachtet. Dieser Ansatz wird im folgenden vorgestellt. Die Darstellung folgt im wesentlichen [Nie93c] und [Nie93b], vgl. auch [GvzG94].

Sei wieder $\mathbb{F}_q$ ein endlicher Körper der Charakteristik p mit $q = p^r$ Elementen und $f \in \mathbb{F}_q[X]$ ein normiertes, *nicht notwendig quadratfreies* Polynom vom Grad $n > 0$, dessen kanonische Faktorisierung

$$f = g_1^{e_1} \cdot \ldots \cdot g_t^{e_t} \tag{1.10}$$

zu berechnen ist. Wir betrachten die Differentialgleichung

$$y^{(p-1)} = -y^p \tag{1.11}$$

über dem Körper der rationalen Funktionen $\mathbb{F}_q(X)$ (dabei bezeichne $y^{(k)}$ die k-te formale Ableitung der rationalen Funktion y nach der Unbestimmten X). Speziell interessieren wir uns für diejenigen Lösungen y mit *festem* Nenner f.

Definition 1.9 *Die Menge*

$$\mathcal{N} := \left\{ h \in \mathbb{F}_q[X];\ \left(\frac{h}{f}\right)^{(p-1)} = -\frac{h^p}{f^p} \right\} \tag{1.12}$$

$$= \left\{ h \in \mathbb{F}_q[X];\ f^p \left(\frac{h}{f}\right)^{(p-1)} = -h^p \right\} \tag{1.13}$$

heißt der (zu f gehörige) Niederreiter-Raum *über* $\mathbb{F}_p$.

Lemma 1.10 *Sei f wie in (1.10).*

i. $h \in \mathcal{N} \Rightarrow \deg h < n$

ii. $\mathcal{N}$ *ist ein t-dimensionaler $\mathbb{F}_p$-Unterraum von* $\mathbb{F}_q[X]$

iii.

$$\mathcal{N} = \left\{ c_1 \frac{f g_1'}{g_1} + \ldots + c_t \frac{f g_t'}{g_t};\ c_1, \ldots, c_t \in \mathbb{F}_p \right\} \tag{1.14}$$

iv. Für $h = \sum_{1 \le i \le t} c_i \frac{f g_i{}'}{g_i} \in \mathcal{N}$ *ist*

$$ggT(f, h) = \prod_{1 \le i \le t} g_i^{e_i - 1} \cdot \prod_{c_i = 0} g_i = \frac{f}{f_{\boxtimes}} \prod_{c_i = 0} g_i = \frac{f}{\prod_{c_i \ne 0} g_i}$$

Beweis:

i. Wir definieren den *Grad einer rationalen Funktion* in der üblichen Weise als $\deg \frac{g}{h} := \deg g - \deg h$, wobei $\deg 0 := -\infty$ gesetzt wird. Damit gilt offenbar $\deg \left(\frac{h}{f}\right)' \le \deg \frac{h}{f} - 1$, woraus per Induktion $\deg \left(\frac{h}{f}\right)^{(k)} \le \deg \frac{h}{f} - k$ für alle $k \in \mathbb{N}$ folgt. Ein Gradvergleich in (1.12) ergibt

$$p(\deg h - \deg f) \le \deg h - \deg f - (p-1)$$

was nur für $\deg h < \deg f = n$ der Fall sein kann.

ii. $\mathcal{N}$ ist der Kern des $\mathbb{F}_p$-linearen Operators

$$\nu' : \begin{cases} \mathbb{F}_q[X]_n & \longrightarrow \quad \mathbb{F}_q[X] \\ h & \mapsto \quad f^p \left(\frac{h}{f}\right)^{(p-1)} + h^p \end{cases} \tag{1.15}$$

und somit ein $\mathbb{F}_p$-Unterraum des nr-dimensionalen $\mathbb{F}_p$-Vektorraums $\mathbb{F}_q[X]_n := \{h \in \mathbb{F}_q[X];\ \deg h < n\}$. Die Dimension von $\mathcal{N}$ ergibt sich aus dem Beweis von iii.

iii. Wir zeigen zunächst $\frac{fg_i'}{g_i} \in \mathcal{N}$ für $1 \le i \le t$.

In einem geeigneten Erweiterungskörper von $\mathbb{F}_q$ zerfällt g_i in Linearfaktoren. Sei W_i die Menge der Wurzeln von g_i in diesem Körper, d.h. $g_i = \prod_{w \in W_i}(X-w)$. Dann gilt:

$$\begin{aligned} \frac{g_i'}{g_i} &= (\log g_i)' = \left(\log \prod_{w\in W_i}(X-w)\right)' = \left(\sum_{w\in W_i}\log(X-w)\right)' \\ &= \sum_{w\in W_i}(\log(X-w))' = \sum_{w\in W_i}\frac{1}{X-w} \end{aligned}$$

Damit erhalten wir

$$\begin{aligned} \left(\frac{g_i'}{g_i}\right)^{(p-1)} &= \left(\sum_{w\in W_i}\frac{1}{X-w}\right)^{(p-1)} = \sum_{w\in W_i}\left(\frac{1}{X-w}\right)^{(p-1)} \\ &= \sum_{w\in W_i}(-1)^{p-1}(p-1)!\frac{1}{(X-w)^p} \\ &= \sum_{w\in W_i}(-1)^p\frac{1}{(X-w)^p} = -\left(\sum_{w\in W_i}\frac{1}{X-w}\right)^p \\ &= -\left(\frac{g_i'}{g_i}\right)^p \end{aligned}$$

Die vierte Gleichung ergibt sich aus der Wilson'schen Identität $(p-1)! \equiv -1 \pmod p$ ([IR82], Corollary zu Prop. 4.1.1). Mit $h := f\frac{g_i'}{g_i}$ ist $\left(\frac{h}{f}\right)^{(p-1)} = -\frac{h^p}{f^p}$.

Die $\frac{fg_i'}{g_i}$ sind linear unabhängig, denn

$$\begin{aligned} \sum_{1\le i\le t} c_i\frac{fg_i'}{g_i} = 0 \quad &\Rightarrow \quad g_i^{e_i} \text{ teilt } \left(\sum_{1\le i\le t} c_i\frac{fg_i'}{g_i}\right) \quad (1 \le i \le t) \\ &\Rightarrow \quad c_i = 0 \quad (1\le i\le t) \end{aligned}$$

Für den Beweisabschluß genügt es zu zeigen, daß *jede* Lösung $y = \frac{h}{f}$ von (1.11) von der Form (1.14) ist. Wir folgen dazu [Nie93a], Beweis zu Theorem 2. Nach i. ist $\deg h < \deg f$. Sei W die Menge aller Wurzeln von f in seinem Zerfällungskörper E über $\mathbb{F}_q$, und bezeichne $e(w)$ die Vielfachheit der Nullstelle $w \in W$ von f (d.h. $W = \bigsqcup_{1 \leq i \leq t} W_i$ (disjunkte Vereinigung) und $e(w) = e_i$ für $w \in W_i$). Die Partialbruchzerlegung von y über $E(X)$ ergibt sich als

$$y = \frac{h}{f} = \sum_{w \in W} \sum_{1 \leq j \leq e(w)} \frac{\gamma(w,j)}{(X-w)^j}$$

mit geeigneten $\gamma(w,j) \in E$. Nach $(p-1)$-facher Differentiation erhalten wir

$$y^{(p-1)} = \sum_{w \in W} \sum_{1 \leq j \leq e(w)} (-1)^{p-1} \cdot j \cdot (j+1) \cdot \ldots \cdot (j+p-2) \frac{\gamma(w,j)}{(X-w)^{j+(p-1)}} \quad (1.16)$$

Die rechte Seite der Differentialgleichung (1.11) läßt sich analog schreiben als

$$-y^p = \sum_{w \in W} \sum_{1 \leq j \leq e(w)} -\frac{\gamma(w,j)^p}{(X-w)^{jp}} \quad (1.17)$$

Wegen der Eindeutigkeit der Partialbruchzerlegung von $y^{(p-1)} = -y^p$ müssen in den Nennern derjenigen Summanden von (1.16) und (1.17), deren Koeffizienten nicht verschwinden, genau dieselben Exponenten auftreten. Aus Gradgründen kann das aber nur für $\gamma(w,j) = 0$ $(j \geq 2)$ gelten, d.h. mit $\gamma(w) := \gamma(w,1)$ ergibt sich

$$y = \sum_{w \in W} \frac{\gamma(w)}{(X-w)^i} \quad (1.18)$$

und durch Gleichsetzen von (1.16) und (1.17)

$$\begin{aligned} y^{(p-1)} &= \sum_{w \in W} (-1)^{p-1}(p-1)! \frac{\gamma(w)}{(X-w)^p} && (1.19) \\ &= \sum_{w \in W} -\frac{\gamma(w)}{(X-w)^p} \\ &= \sum_{w \in W} -\frac{\gamma(w)^p}{(X-w)^p} && (1.20) \\ &= -y^p \end{aligned}$$

Daraus folgt — wieder wegen der Eindeutigkeit der Partialbruchzerlegung von $y^{(p-1)}$ — $\gamma(w) \in \mathbb{F}_p$ für alle $w \in W$.

Jeder Galois-Automorphismus von E über $\mathbb{F}_q$ permutiert jeweils die zu *einem* W_i gehörigen Summanden in der Partialbruchzerlegung (1.18) von y untereinander. Für je zwei $w, \overline{w} \in W_i$ sind demnach $\gamma(w)$ und $\gamma(\overline{w})$ konjugiert, also wegen $\gamma(w), \gamma(\overline{w}) \in \mathbb{F}_p \subseteq \mathbb{F}_q$ sogar gleich. Wir können alle Summanden in (1.18), die zu demselben W_i gehören, zusammenfassen, den allen $w \in W_i$ gemeinsamen Koeffizienten $c_i := \gamma(w)$ ausklammern und erhalten

$$y = \frac{h}{f} = \sum_{1 \leq i \leq t} c_i \sum_{w \in W_i} \frac{1}{X - w} = \sum_{1 \leq i \leq t} c_i \frac{g_i'}{g_i}$$

iv. Offenbar gilt $\frac{f}{f_\boxtimes} | \frac{f g_i'}{g_i}$ für alle i und somit $\frac{f}{f_\boxtimes} | h$. Weiterhin ist

$$\frac{f_\boxtimes}{f} h = \sum_{1 \leq i \leq t} c_i \frac{f_\boxtimes g_i'}{g_i}$$

Jeder Summand auf der rechten Seite – bis auf möglicherweise den i-ten – ist durch g_i teilbar. Der i-te Summand ist aber genau dann durch g_i teilbar, wenn $c_i = 0$ ist.

□

Bemerkung Der Niederreiter-Raum $\mathcal{N}$ ist *keine* $\mathbb{F}_p$-Algebra, da der Operator ν' aus dem Beweis von Lemma 1.10 zwar $\mathbb{F}_p$-linear, aber nicht multiplikativ ist!

Wir betrachten den Operator ν' aus (1.15) etwas genauer. Aus Gleichung (1.19) ist ersichtlich, daß Zähler und Nenner von $y^{(p-1)}$ für eine rationale Funktion $y \in \mathbb{F}_q(X)$ Polynome in X^p sind. Damit bildet der lineare Operator ν' ein Polynom $h \in \mathbb{F}_q[X]$ vom Grad $< n$ auf ein Polynom vom Grad $< pn$ ab, in dem nur Potenzen von X^p auftreten. Bezeichnen wir die Menge aller solchen Polynome mit $\mathbb{F}_q[X^p]_n$, so kann ν' als eine lineare Abbildung $\mathbb{F}_q[X]_n \longrightarrow \mathbb{F}_q[X^p]_n$ aufgefaßt werden, wobei Definitions- und Wertebereich dieselbe Dimension nr über $\mathbb{F}_p$ haben. Das Kompositum $\nu := \lambda^{-1} \circ \nu'$ von ν' mit dem Inversen des kanonischen $\mathbb{F}_p$-Isomorphismus

$$\lambda : \begin{cases} \mathbb{F}_q[X]_n & \longrightarrow & \mathbb{F}_q[X^p]_n \\ X & \mapsto & X^p \end{cases}$$

ist somit ein $\mathbb{F}_p$-Endomorphismus von $\mathbb{F}_q[X]_n$ mit Kern $\mathcal{N}$.

Wir beschreiben nun eine an Algorithmus 1.4 angelehnte, deterministische Version des Niederreiter-Algorithmus für den Fall eines *quadratfreien* Polynoms f.

Algorithmus 1.11 (Niederreiter)

{ *input:* $f \in \mathbb{F}_q[X]$ *normiert, quadratfrei,* $\deg f = n$ }

{ *output:* $g_1, \dots, g_t$*, die irreduziblen Faktoren von* f }

I. (setup) *Berechne die Koeffizienten* n_{ij} *der zum linearen Operator* ν *gehörenden* $nr \times nr$*-Matrix* $N = (n_{ij})_{1 \leq i,j \leq nr}$ *bezüglich der geordneten („Polynom“)-Basis*

$$\{1, \alpha, \dots, \alpha^{r-1}, X, \alpha X, \dots, \alpha^{r-1}X, \dots, X^{n-1}, \alpha X^{n-1}, \dots, \alpha^{r-1}X^{n-1}\} \tag{1.21}$$

von $\mathbb{F}_q[X]_n$ *über* $\mathbb{F}_p$*. Hierbei bezeichne* α *ein beliebiges, fest gewähltes Element, welches die Körpererweiterung* $\mathbb{F}_q/\mathbb{F}_p$ *erzeugt.*

II. *Berechne mittels Gauß-Elimination den Rang* $nr - t$ *von* N *und eine Basis* $\{h_1, \dots, h_t\}$ *von* $\ker N = \{h \in \mathbb{F}_q[X]_n; Nh = 0\}$ *(o.E. sei* $h_1 = f'$*).*

III. *Setze factorbase* $:= \{f\}$*.*

IV. *Wiederhole die Schritte V. und VI. für alle* $a \in \mathbb{F}_p$ *und* $(1 < k \leq t)$ *so lange, bis* $\#$*factorbase* $= t$*:*

V. *Setze oldfactorbase* $:=$ *factorbase und factorbase* $:= \emptyset$*.*

VI. (split) *Für alle* $b \in$ *oldfactorbase berechne* $g := ggT(h_k - af', b)$*.*
Falls $g \neq 1, b$*, setze factorbase* $:=$ *factorbase* $\cup \{g, \frac{b}{g}\}$*,*
andernfalls factorbase $:=$ *factorbase* $\cup \{b\}$*.*

VII. *output factorbase.*

Bemerkung 1.12 (Korrektheit und Komplexität)

- Ein Polynom $h \in \mathcal{N}$ läßt sich nach Lemma 1.10 eindeutig als Linearkombination

$$h = a_1 \frac{fg_1}{g_1'} + \ldots + a_t \frac{fg_t}{g_t'} \tag{1.22}$$

mit $a_1, \dots, a_t \in \mathbb{F}_p$ darstellen. Wegen f quadratfrei gilt insbesondere $h = f' \in \mathcal{N}$ mit $a_1 = \ldots = a_t = 1$, denn

$$\frac{f'}{f} = (\log f)' = (\log \prod_{1 \leq i \leq t} g_i)' = \sum_{1 \leq i \leq t} (\log g_i)' = \sum_{1 \leq i \leq t} \frac{g_i'}{g_i} \tag{1.23}$$

Also ist für $a \in \mathbb{F}_p$ und $h \in \mathcal{N}$ wie in (1.22) $h - af' = \sum_{1 \le i \le t}(a_i - a)\frac{fg_i'}{g_i}$ und damit

$$\text{ggT}(f, h - af') = \prod_{1 \le i \le i;\ a_i = a} g_i \tag{1.24}$$

Diese Zerlegung ist genau dann trivial, wenn h von der Form $h = b \cdot f'$ für ein $b \in \mathbb{F}_p$ ist.

Analog zur Situation beim Berlekamp-Algorithmus läßt sich zeigen, daß jede Basis $\{h_1, \ldots, h_t\}$ von $\mathcal{N}$ eine „separierende Menge" in dem Sinne ist, daß durch die Schritte IV. bis VI. von Algorithmus 1.11 je zwei irreduzible Faktoren $g_i \neq g_j$ von f getrennt werden (vgl. [GvzG94] p. 11).

- Niederreiter reduziert in [Nie94] die Berechnung der Matrix N auf die Berechnung der Berlekamp-Matrix für den p-Frobenius-Morphismus von R. Die „setup"-Kosten für die Matrix N in Schritt I. betragen danach $\mathcal{O}((\log p + n)M(n))$ Körperoperationen in $\mathbb{F}_q$ für die Aufstellung der Berlekamp-Matrix, sowie zusätzlich $O(n^3)$ $\mathbb{F}_q$-Operationen und $\mathcal{O}(n^2r^3)$ $\mathbb{F}_p$-Operationen.

- Die Berechnung einer Basis von $\ker N = \mathcal{N}$ in Schritt II. benötigt $\mathcal{O}(n^3r^3)$ Operationen in $\mathbb{F}_p$.

- Die Kosten für *einen* „splitting"-Schritt VI. betragen analog zum Berlekamp-Algorithmus $\mathcal{O}(n^2)$ bzw. $\mathcal{O}(M(n)\log n)$ Körperoperationen in $\mathbb{F}_q$ bei Verwendung klassischer bzw. schneller Polynomarithmetik. In $\mathbb{F}_p$-Operationen gemessen sind das $\mathcal{O}(n^2r^2)$ bzw. $\mathcal{O}(M(n)\log n \cdot M(r)\log r)$.

- Damit ergibt sich eine Gesamtlaufzeit von Algorithmus 1.11 von $\mathcal{O}(n^3r^3 + p \cdot tn^2r^2) = \mathcal{O}(n^3r^2(r+p))$ $\mathbb{F}_p$-Operationen mit klassischer Arithmetik bzw.

  $$\mathcal{O}(n^3r^3 + p \cdot tM(n)\log n \cdot M(r)\log r) = \mathcal{O}(n^3r^3 + p \cdot nM(n)\log n \cdot M(r)\log r)$$

  mit schnelleren Techniken.

Bemerkung 1.13

1. Der vorgestellte Algorithmus zeigt gewisse Ähnlichkeiten mit dem Berlekamp-Algorithmus: Zunächst wird eine Basis des Kerns eines linearen Operators durch Elimination einer geeigneten Matrix berechnet, aus der anschließend durch sukzessive Verfeinerung die vollständige Faktorisierung von f gewonnen wird. Allerdings ist der Operator ν nur $\mathbb{F}_p$-linear, aber nicht $\mathbb{F}_q$-linear wie der Operator β beim Berlekamp-Algorithmus. Wie in

Bemerkung 1.7 angedeutet, kann Berlekamps Algorithmus dahingehend modifiziert werden, daß anstelle der relativen die absolute Berlekamp-Algebra und anstelle der $\mathbb{F}_q$-linearen Abbildung β eine entsprechende, lediglich $\mathbb{F}_p$-lineare Abbildung verwendet wird. Diese Variante weist noch deutlichere Ähnlichkeiten mit Algorithmus 1.11 auf und hat dieselbe asymptotische Laufzeit.

Niederreiter gibt in [Nie93b] aber auch eine „relative" Abwandlung seines Verfahrens an, in der an die Stelle des $\mathbb{F}_p$-linearen Operators ν ein $\mathbb{F}_q$-linearer Operator tritt. Dabei wird die gewöhnliche k-te formale Ableitung ersetzt durch die sog. *Hasse-Teichmüller-Ableitung* $H^{(k)}$. Diese hat gegenüber der ersteren den Vorteil, daß sie in Charakteristik p nicht für $k \geq p$ identisch verschwindet. Statt der Differentialgleichung (1.11) wird die Gleichung

$$H^{(q-1)}y + y^q = 0$$

über $\mathbb{F}_q(X)$ betrachtet. Das darauf basierende Verfahren eliminiert eine wesentlich kleinere Matrix (nämlich der Dimension n über $\mathbb{F}_q$ statt Dimension nr über $\mathbb{F}_p$). Die Matrix N in Algorithmus 1.11 ist jedoch für kleine p (insbesondere $p = 2$) besser handhabbar.

2. Algorithmus 1.11 unterscheidet sich geringfügig von dem in [Nie93b]: Niederreiter selbst verwendet statt des erzeugenden Elements α von $\mathbb{F}_q/\mathbb{F}_p$ und der Basis (1.21) eine Normalbasis $\{\eta, \eta^p, \ldots, \eta^{p^{r-1}}\}$ von $\mathbb{F}_q/\mathbb{F}_p$ und entsprechend als $\mathbb{F}_p$-Basis von $\mathbb{F}_q[X]_n$ die Basis

$$\{\eta, \eta^p, \ldots, \eta^{p^{r-1}}, \eta X, \eta^p X, \ldots, \eta^{p^{r-1}} X, \ldots, \eta X^{n-1}, \eta^p X^{n-1}, \ldots, \eta^{p^{r-1}} X^{n-1}\}$$

Die Darstellung von ν bezüglich dieser Basis erleichtert — bei Wahl einer geeigneten Normalbasis — die Aufstellung der zugehörigen Matrix, bringt allerdings für die asymptotische Laufzeit des Verfahrens keinen Gewinn.

3. Niederreiters Algorithmus läßt sich so modifizieren, daß er — ohne vorherige quadratfreie Zerlegung! — auch für nicht quadratfreie Polynome f die vollständige Faktorisierung liefert (vgl. dazu [GvzG94], [Nie93b]).

Das folgende Resultat stammt aus [GvzG94], Thm. 2.1(v), und gestattet es, eine probabilistische „large prime"-Version von Niederreiters Algorithmus anzugeben, die die Berechnung der vollständigen Faktorisierung von f aus einer Basis von $\mathcal{N}$ für den Fall $p \neq 2$ wesentlich beschleunigt.

Lemma 1.14 *Sei S die Menge der Quadrate in $\mathbb{F}_p^*$, $p \neq 2$. Für $h = \sum_{1 \le i \le t} a_i \frac{fg_i'}{g_i}$ in $\mathcal{N}$ ($a_i \in \mathbb{F}_p$ für $1 \le i \le t$) gilt*

$$ggT(f, h^{\frac{p-1}{2}} - (f')^{\frac{p-1}{2}}) = \prod_{1 \le i \le t;\, a_i \in S} g_i$$

Beweis: Wegen $\frac{fg_i'}{g_i} \cdot \frac{fg_j'}{g_j} \equiv 0 \pmod f$ für $i \neq j$ gilt

$$h^{\frac{p-1}{2}} - (f')'^{\frac{p-1}{2}} \equiv \sum_{1 \le i \le t} (a_i^{\frac{p-1}{2}} - 1) \cdot \left(\frac{fg_i'}{g_i}\right)^{\frac{p-1}{2}} \pmod f$$

Die rechte Seite ist jedoch genau dann durch g_i teilbar, wenn $a_i^{\frac{p-1}{2}} = 1$, d.h. a_i ein Quadrat in $\mathbb{F}_p$ ist. $\square$

Algorithmus 1.15 („large prime“ Niederreiter)

{ *input, output und Schritte I. und II. wie in Algorithmus 1.11* }

III. *Setze factorbase* $:= \{f\}$.

IV. *Wiederhole die Schritte V. bis VII. so lange, bis #factorbase* $= t$:

V. *Wähle zufällig* $\lambda_1, \ldots, \lambda_t \in \mathbb{F}_p$ *und setze* $h := \sum_{1 \le i \le t} \lambda_i h_i$.

VI. *Setze oldfactorbase := factorbase und factorbase* $:= \emptyset$.

VII. (split) *Für alle* $b \in$ *oldfactorbase berechne* $f_1 := ggT(h, b)$, $f_2 := ggT((h^{\frac{p-1}{2}} - (f')^{\frac{p-1}{2}}) \bmod f, b)$ *und* $f_3 := \frac{b}{f_1 f_2}$ *und setze factorbase := factorbase* $\cup (\{f_1, f_2, f_3\} \setminus \{1\})$.

VIII. *output factorbase.*

Die erwartete Laufzeit von Algorithmus 1.15 ergibt sich analog zu der des „large prime“ Berlekamp-Verfahrens. Sie beträgt

$$\mathcal{O}(n^3 r^3 + (\log p + 2) n^2 \log t \cdot r^2) = \mathcal{O}(n^2 r^2 (nr + \log p \cdot \log n))$$

Körperoperationen in $\mathbb{F}_p$ im klassischen Fall bzw.

$$\begin{aligned} &\mathcal{O}(n^3 r^3 + (\log p + 2\log n) M(n) \log t \cdot M(r) \log r) \\ &= \mathcal{O}(n^3 r^3 + (\log p + \log n) M(n) \log n \cdot M(r) \log r) \end{aligned}$$

$\mathbb{F}_p$-Operationen mit schneller Polynomarithmetik.

1.4 Ein Vergleich von Berlekamps und Niederreiters Algorithmen

Im vorherigen Abschnitt wurden bereits gewisse Parallelen zwischen den Faktorisierungsverfahren von Berlekamp und Niederreiter angedeutet. Der Niederreiter-Raum $\mathcal{N}$ und die absolute Berlekamp-Algebra $\overline{\mathcal{B}}$ haben dieselbe Dimension t über $\mathbb{F}_p$ (vgl. Bem. 1.7) und sind daher als $\mathbb{F}_p$-Vektorräume isomorph. Verschiedene Autoren haben nach der Veröffentlichung von Niederreiters Algorithmus in [Nie93c] einen „kanonischen" Isomorphismus zwischen $\overline{\mathcal{B}}$ und $\mathcal{N}$ angegeben. Im folgenden ist $f \in \mathbb{F}_q[X]$ ein normiertes, quadratfreies Polynom vom Grad n. Dann ist $\mathrm{ggT}(f, f') = 1$ und damit $f' + (f)$ eine Einheit in R. Sei $s \in \mathbb{F}_q[X]$ mit $sf' \equiv 1 \pmod{f}$, das Inverse von f' modulo f.

Satz 1.16 (Fleischmann / Miller ([Fle93], [Mil92])) *Die Abbildung*

$$\psi : \begin{cases} \mathbb{F}_q[X]_n & \longrightarrow \quad R = \mathbb{F}_q[X]/(f) \\ h & \mapsto \quad hs + (f) \end{cases} \tag{1.25}$$

ist ein $\mathbb{F}_p$-Isomorphismus und induziert einen $\mathbb{F}_p$-Isomorphismus zwischen $\mathcal{N} \subseteq \mathbb{F}_q[X]_n$ und $\overline{\mathcal{B}} \subseteq R$. Genauer gilt:

$$\psi\left(\frac{fg_i'}{g_i}\right) = b_i \tag{1.26}$$

wobei $b_1, \ldots, b_t$ die kanonische Basis (1.8) von $\overline{\mathcal{B}}$ bezeichnet.

Beweis: Die $\mathbb{F}_p$-Linearität von ψ ist offensichtlich. Die Abbildung

$$\psi^{-1} : h + (f) \mapsto hf' \bmod f$$

ist die Umkehrabbildung von ψ und damit ψ bijektiv. Es bleibt noch die Aussage (1.26) zu zeigen. Aus (1.23) folgt $f' \equiv \frac{fg_j'}{g_j} \pmod{g_j}$ $(1 \leq j \leq t)$. Damit gilt:

$$\begin{aligned} \psi\left(\frac{fg_i'}{g_i}\right) &= s\frac{fg_i'}{g_i} + (f) = \frac{f}{f'} \cdot \frac{g_i'}{g_i} + (f) \\ &\equiv \frac{g_j}{g_j'} \cdot \frac{g_i'}{g_i} \pmod{g_j} \\ &\equiv \begin{cases} 1 \pmod{g_j} & (j = i) \\ 0 \pmod{g_j} & (j \neq i) \end{cases} \end{aligned}$$

□

Der Isomorphismus ψ kann auch dazu herangezogen werden, die im letzten Abschnitt bewiesenen Eigenschaften von $\mathcal{N}$ aus denen von $\overline{\mathcal{B}}$ abzuleiten (vgl. [GvzG94]).

Bemerkung 1.17

1. Wie bereits in Bemerkung 1.13 angedeutet, kann der Niederreiter-Algorithmus so modifiziert werden, daß er auf beliebige, nicht notwendig quadratfreie Polynome f anwendbar ist. Ob dies jedoch einen Laufzeitgewinn gegenüber einer vorherigen quadratfreien Zerlegung bringt, ist keineswegs klar: Die Endprodukte dieser Zerlegung haben unter Umständen einen viel kleineren Grad als das ursprüngliche Polynom. Es ist also zwischen den Kosten für eine Anwendung des Niederreiter-Verfahrens auf das gesamte Polynom f einerseits und den Kosten für eine quadratfreie Zerlegung von f und die anschließende Anwendung des Algorithmus auf quadratfreie Faktoren von f kleineren Grades andererseits abzuwägen.

   Ebensowenig ist klar, ob diese Eigenschaft von Niederreiters Verfahren einen Vorteil gegenüber dem Berlekamp-Algorithmus bietet. Denn wie in Bemerkung 1.8 dargestellt, kann auch letzterer auf beliebige Polynome f angewendet werden. Das Verfahren liefert dann allerdings nicht die vollständige Faktorisierung, sondern lediglich eine Zerlegung in Primpotenzen, die ggf. anschließend noch quadratfrei zerlegt werden müssen.

2. Da die Aufstellung der Matrix N des Niederreiter-Verfahrens die vorherige Berechnung der Berlekamp-Matrix erfordert, ist der Algorithmus von Niederreiter in diesem Punkt (zumindest bei großer Charakteristik p — s.u.) gegenüber dem von Berlekamp im Nachteil. Niederreiter merkt allerdings in [Nie93b] an, daß sein Verfahren einen Vorteil bietet, wenn sehr viele Polynome gleichen Grades zu faktorisieren sind. In diesem Fall kann mit Hilfe der Quotientenregel (relativ ineffizient) eine „generische“ Matrix vorausberechnet werden, in der die Koeffizienten des Polynoms als Unbekannte auftauchen. Die eigentliche Matrix N kann für jedes zu faktorisierende Polynom f aus der generischen Matrix durch Einsetzen der tatsächlichen Koeffizienten von f gewonnen werden.

3. Niederreiters Algorithmus erweist sich über Körpern der Charakteristik 2 als besonders vorteilhaft. In diesem Fall hat die Matrix N eine einfache Form; eine vorherige Berechnung der Berlekamp-Matrix entfällt. Ist das zu faktorisierende Polynom f außerdem „dünn“ (d.h. sehr viele Koeffizienten sind $= 0$), dann läßt sich zeigen, daß sich diese Eigenschaft auch auf die Matrix N überträgt (vgl. [Nie93b]), was beim Berlekamp-Algorithmus nicht notwendigerweise der Fall ist.

4. Der letzte Schritt in beiden Faktorisierungsalgorithmen, die Berechnung der vollständigen Faktorisierung aus einer Basis des Nullraums von $\overline{B}$ (vgl. Bem. 1.7) bzw. N, gestaltet sich in Charakteristik 2 ebenfalls sehr einfach. Da als Koeffizienten a_i $(1 \leq i \leq t)$ in (1.6) bzw. (1.24) nur die beiden Elemente $0, 1 \in \mathbb{F}_2$ auftreten, liefert $\mathrm{ggT}(f, h)$ für *jedes* Element $h \in \overline{\mathcal{B}}$ bzw. $h \in \mathcal{N}$ mit $h \neq 0, 1$ bzw. $h \neq 0, f'$ einen nichttrivialen Faktor von f. Der „$\frac{p-1}{2}$-Trick“ ist in diesem Fall gar nicht nötig.

Kapitel 2

Lineare Algebra mit „black box“-Matrizen

2.1 Wiedemanns Algorithmus

Die im vorigen Kapitel vorgestellten Verfahren führen das Faktorisierungsproblem auf ein Problem der linearen Algebra zurück: Die Elimination einer Matrix zur Bestimmung einer Basis des Nullraums. Der gravierende Nachteil daran ist, daß die klassischen Eliminationsalgorithmen (wie z.B. Gauß) alle kubische Laufzeit haben, d.h. sie benötigen $\mathcal{O}(n^3)$ Körperoperationen für eine $n \times n$-Matrix. Theoretisch existieren zwar asymptotisch schnellere Verfahren mit einer Laufzeit von $\mathcal{O}(n^\omega)$ für $\omega < 3$; sie sind jedoch meist nicht praktikabel.

Im Vergleich dazu betragen die Kosten für Polynomarithmetik (Multiplikation, Division, ggT-Berechnung) auf Polynomen vom Grad n im klassischen Fall $\mathcal{O}(n^2)$ Körperoperationen und bis zu $\mathcal{O}(n(\log n)^2 \log\log n)$ bei Verwendung schnellerer Techniken. Daher erscheint es wünschenswert, ein Faktorisierungsverfahren zu finden, das *ohne* klassische lineare Algebra auskommt und statt dessen nicht zu viele Polynomoperationen verwendet.

Wiedemann präsentierte im Jahre 1986 ein probabilistisches „Las Vegas“-Verfahren zur Lösung linearer Gleichungssysteme, welches das Problem darauf reduziert, das Minimalpolynom der Koeffizientenmatrix zu berechnen (vgl. [Wie86]). Dazu braucht die Matrix selbst nicht koeffizientenweise bekannt zu sein, es muß lediglich möglich sein, die Matrix mit einem beliebigen Vektor zu multiplizieren (*„black box“*-Repräsentation). Sind die Kosten dafür „klein“ (d.h. eine Anwendung der $n \times n$-Matrix auf einen n-Vektor benötigt $\mathcal{O}(n^e)$ Körperoperationen mit $e < 2$), dann erweist sich — zumindest asymptotisch — Wiedemanns Methode gegenüber der Elimination als überlegen.

Anwendungsmöglichkeiten dafür ergeben sich z.B., wenn die Matrix „dünn“ ist (d.h. nur $\mathcal{O}(n^e)$ Koeffizienten sind $\neq 0$ für ein $e < 2$), oder — und das ist der für uns interessante Fall — wenn eine Anwendung der Matrix auf eine feste Anzahl von Polynomoperationen zurückgeführt werden kann. Die „black box“-Repräsentation der Matrix hat auch den Vorteil, daß kein Speicher für die $\mathcal{O}(n^2)$ Koeffizienten gebraucht wird.

Bevor wir die Idee von Wiedemann vorstellen, benötigen wir noch einige Begriffe aus der Theorie der linear rekurrenten Folgen. Sei $\mathbb{F}$ ein beliebiger Körper und V ein $\mathbb{F}$-Vektorraum.

Definition 2.1 *Eine Folge $a := (a_i)_{i\geq 0} = (a_0, a_1, a_2, \ldots) \subseteq V$ heißt eine (über $\mathbb{F}$)* linear rekurrente Folge, *wenn es ein $n \geq 0$ und $c_0, \ldots, c_n \in \mathbb{F}$ mit $c_n \neq 0$ gibt, so daß*

$$\sum_{0\leq j\leq n} c_j a_{i+j} = 0 \textit{ für alle } i \geq 0$$

Das Polynom $c := \sum_{0\leq j\leq n} c_j X^j \in \mathbb{F}[X]$ wird ein generierendes Polynom, charakteristisches Polynom *oder auch* Rückkopplungspolynom *für a genannt (vgl. [Ber68], [Mas69]).*

Die Menge V^∞ aller (abzählbaren) Folgen $a \subseteq V$ ist ein unendlich-dimensionaler $\mathbb{F}$-Vektorraum. Die Einführung des äußeren Produkts „$\cdot$“: $\mathbb{F}[X] \times V^\infty \longrightarrow V^\infty$ durch die Festlegungen

$$\begin{aligned} c \cdot (a_i)_{i\geq 0} &:= (ca_i)_{i\geq 0} \quad (c \in \mathbb{F}) \\ X \cdot (a_i)_{i\geq 0} &:= (a_{i+1})_{i\geq 0} \end{aligned}$$

macht V^∞ zu einem $\mathbb{F}[X]$-Modul. In diesem Sinne ist eine Folge a genau dann linear rekurrent, wenn es ein Polynom $f \in \mathbb{F}[X]\backslash\{0\}$ mit $f \cdot a = 0$ gibt, wobei $0 := (0, 0, 0, \ldots)$ die Nullfolge ist.

Für eine Folge $a \in V^\infty$ bezeichnen wir mit

$$\mathrm{Ann}(a) := \{f \in \mathbb{F}[X];\ f \cdot a = 0\}$$

die Menge aller Polynome, die die Folge a generieren, genannt der *Annullator* von a. $\mathrm{Ann}(a)$ ist ein (Haupt-)Ideal von $\mathbb{F}[X]$. Falls a linear rekurrent ist, d.h. $\mathrm{Ann}(a) \neq \{0\}$, existiert also ein eindeutiges normiertes Polynom $m \in \mathrm{Ann}(a)\backslash\{0\}$ kleinsten Grades mit $\mathrm{Ann}(a) = (m)$. Dieses Polynom m heißt das *Minimalpolynom von a* und hat die Eigenschaft, daß es jedes generierende Polynom von a teilt. Der Grad von m wird als die *Rekursionsordnung* der Folge a bezeichnet.

Beispiel 2.2

1. Die Nullfolge $0 = (0, 0, 0, \ldots)$ hat das Minimalpolynom 1.

2. $V := \mathbb{F}^{n \times n}$. $a := (I, A, A^2, A^3, \ldots)$ ist für jede quadratische Matrix $A \in V$ linear rekurrent. Hier ist
$$f \cdot a = (A^i \cdot f(A))_{i \geq 0}$$
für ein Polynom $f \in \mathbb{F}[X]$ und damit $f \cdot a = 0 \Leftrightarrow f(A) = 0$. Das Minimalpolynom von a ist daher das Minimalpolynom m_A von A und hat nach dem Satz von Cayley-Hamilton höchstens Grad n.

3. $V' := \mathbb{F}^n$. $a' := (b, Ab, A^2b, A^3b, \ldots)$ ist für A wie in 2. und $b \in V'$ ebenfalls linear rekurrent. Die Moduloperation ist hier gegeben durch
$$f \cdot a' = (A^i \cdot f(A)b)_{i \geq 0}$$
und es gilt $f \cdot a' = 0 \Leftrightarrow f(A)b = 0$. Offenbar ist m_A ein charakteristisches Polynom für a'. Das Minimalpolynom $m_{A,b}$ von a' ist also ein Teiler von m_A.

4. $V'' := \mathbb{F}$. $a'' := (u^{\mathbf{T}}b, u^{\mathbf{T}}Ab, u^{\mathbf{T}}A^2b, u^{\mathbf{T}}A^3b, \ldots)$ ist für A, b wie in 3. und $u \in \mathbb{F}^n$ linear rekurrent. Für $f \in \mathbb{F}[X]$ ist
$$f \cdot a'' = (u^{\mathbf{T}}A^i \cdot f(A)b)_{i \geq 0}$$
$m_{A,b}$ ist ein generierendes Polynom für a'' und ein Vielfaches des Minimalpolynoms $m_{u,A,b}$ von a''.

Sei nun $\mathbb{F} = \mathbb{F}_q$ ein endlicher Körper, $A \in \mathbb{F}^{n \times n}$ eine invertierbare Matrix und $b \in \mathbb{F}^n$ ein Vektor. Um das Gleichungssystem $Ax = b$ für $x \in \mathbb{F}^n$ zu lösen, verfährt Wiedemanns Methode wie folgt:

Algorithmus 2.3 (Wiedemann)

{ *input: Dimension $n > 0$, „black box“-Algorithmus für die Anwendung von $A \in \mathbb{F}^{n \times n}$, A invertierbar, auf einen Vektor $\in \mathbb{F}^n$, $b \in \mathbb{F}^n$* }

{ *output: Die Lösung $x := A^{-1}b$* }

I. *Berechne das Minimalpolynom $m_{A,b}$ der Folge $(b, Ab, A^2b, A^3b, \ldots)$.*

II. *Sei $m_{A,b} = \sum_{0 \leq j \leq d} m_j X^j$ mit $d \leq n$, $m_d = 1$ und $m_0 \neq 0$ wegen der Nichtsingularität von A.*
Setze $\overline{m} := \frac{m_{A,b} - m_0}{X}$ und $x := -m_0^{-1} \cdot \overline{m}(A)b$.

III. *output x.*

Die Korrektheit des Verfahrens ist leicht einzusehen:

$$Ax = -m_0^{-1} \cdot A \cdot \overline{m}(A)b = -m_0^{-1}(\underbrace{m_{A,b}(A)b}_{=0} - m_0 b) = b$$

Wir notieren die für eine „black box“-Anwendung der n-dimensionalen Matrix A auf einen beliebigen Vektor $h \in \mathbb{F}^n$ benötigte Anzahl Körperoperationen in $\mathbb{F}$ mit $\mathrm{BB}(n)$. Für die probabilistische Bestimmung des Minimalpolynoms in Schritt I., auf die später genauer eingegangen wird, sind die $\mathcal{O}(n)$ ersten Folgenglieder $b, Ab, A^2b, \ldots$ zu berechnen. Schritt I. kostet damit im wesentlichen $\mathcal{O}(n \cdot \mathrm{BB}(n))$ Operationen, ebenso wie die Berechnung von $\overline{m}(A)b$ in Schritt II. mit einem dem Horner-Schema ähnlichen Verfahren. Wiedemanns Algorithmus zur Lösung von $Ax = b$ hat also eine (erwartete) Gesamtlaufzeit von $\mathcal{O}(n \cdot \mathrm{BB}(n))$ — gegenüber $\mathcal{O}(n^3)$ für die Gauß-Elimination — und ist damit zumindest asymptotisch schneller, falls $\mathrm{BB}(n)$ langsamer als quadratisch wächst.

Wie aber wird das Minimalpolynom von $(b, Ab, A^2b, \ldots)$ berechnet? Für den in Bsp. 2.2 unter 4. dargestellten Fall einer linear rekurrenten Folge von Elementen aus dem Grundkörper $\mathbb{F}$ gibt es ein Standard-Verfahren nach Berlekamp und Massey, welches das kürzeste linear rückgekoppelte Schieberegister findet, das die betreffende Folge erzeugt. Aus diesem Schieberegister kann das Minimalpolynom direkt abgelesen werden. Aber auch der erweiterte Euklidische Algorithmus kann für das Problem herangezogen werden (darauf wird in Abschnitt 3. genauer eingegangen).

Sowohl der Berlekamp-Massey-Algorithmus als auch der Euklidische Algorithmus benötigen die ersten $2n$ Folgenglieder. Beide haben eine Laufzeit von $\mathcal{O}(n^2)$ Körperoperationen, die durch „divide and conquer“ noch auf $\mathcal{O}(M(n)\log n)$ reduziert werden kann.

Wir gehen im folgenden davon aus, daß ein Algorithmus „minpoly“ gegeben ist, der das Minimalpolynom von linear rekurrenten Folgen $a \subseteq \mathbb{F}$ berechnet. Er nimmt als Argument die ersten $2n$ Glieder einer Folge $a \subseteq \mathbb{F}$ von Körperelementen, deren Rekursionsordnung höchstens n beträgt, und berechnet das Minimalpolynom dieser Folge mit einer Laufzeit von $\mathcal{O}(n^2)$.

Wiedemann reduziert die Bestimmung des Minimalpolynoms der Folge $(b, Ab, A^2b, \ldots) \in (\mathbb{F}^n)^\infty$ auf die Berechnung von Minimalpolynomen über $\mathbb{F}^\infty$. Für das Verfahren ist es nicht erforderlich, daß die Matrix A invertierbar ist.

Algorithmus 2.4 (Wiedemann)

{ *input: Dimension* $n > 0$, *„black box“-Algorithmus für die Anwendung von* $A \in \mathbb{F}^{n\times n}$ *auf einen Vektor* $\in \mathbb{F}^n$, $b \in \mathbb{F}^n$ }

{ *output:* $m_{A,b}$, *das Minimalpolynom von* $(b, Ab, A^2b, \ldots)$ }

I. *Setze* $m := 1$; *maxdegree* $:= n$ *und* $h := b$.

II. *Wiederhole die Schritte III. und IV. so lange, bis* $h = 0$:

III. *Wähle zufällig* $u \in \mathbb{F}^n$ *und berechne*
$a_i := u^{\mathbf{T}} A^i h \in \mathbb{F}$ *für* $0 \le i < 2 \cdot maxdegree$.

IV. *Setze* $m_1 := minpoly(a_0, \ldots, a_{2 \cdot maxdegree - 1})$,
$m := m \cdot m_1$,
$maxdegree := maxdegree - \deg m_1$,
$h := m_1(A)h$.

V. *output* m.

Korrektheitsbeweis: Wir zeigen per Induktion über die Anzahl Durchläufe der Schleife II., daß folgende Schleifeninvarianten gelten:

i. m ist ein Teiler von $m_{A,b}$

ii. $\deg m + \text{maxdegree} = n$

iii. $h = m(A)b$

Vor Eintritt in die Schleife II. sind alle Bedingungen offensichtlich erfüllt. Nach Induktionsvoraussetzung ist in Schritt IV. $m_{A,b} = m \cdot d$ für ein geeignetes Polynom $d \in \mathbb{F}[X]$. Damit haben wir

$$0 = m_{A,b}(A)b = d(A)m(A)b = d(A)h$$

Wegen $h \neq 0$ ist d ein charakteristisches Polynom für $(h, Ah, A^2h, \ldots)$, also ein Vielfaches von $m_{A,h}$*. Für den Grad des Minimalpolynoms $m_{u,A,h}$ gilt somit

$$\begin{aligned} \deg m_{u,A,h} &\le \deg m_{A,h} \le \deg d = \deg m_{A,b} - \deg m \\ &\le n - \deg m = \text{maxdegree (per Induktion)} \end{aligned}$$

In Schritt IV. wird deswegen korrekt $m_1 = m_{u,A,h}$ berechnet. Zusammengenommen ergibt das $m_1 | m_{A,h} | d$.

Nun folgt der Induktionsschluß leicht aus den Induktionsannahmen und den Zuweisungen in Schritt VI. Wir bezeichnen mit $m_v, h_v, \text{maxdegree}_v$ bzw. m_n, h_n, maxdegree_n die Werte der drei Variablen vor bzw. nach diesem Schritt. An den mit einem „!“ gekennzeichneten Stellen geht die Induktionshypothese ein.

*wegen der Minimalität von $m_{A,b}$ gilt sogar $d = m_{A,h}$

i. $m_1|d \Rightarrow m_n = (m_v \cdot m_1)|(m_v \cdot d) \stackrel{!}{=} m_{A,b}$

ii. $\deg m_n + \mathrm{maxdegree}_n = (\deg m_v + \deg m_1) + (\mathrm{maxdegree}_v - \deg m_1) \stackrel{!}{=} n$

iii. $h_n = m_1(A)h_v \stackrel{!}{=} m_1(A)m_v(A)b = m_n(A)b$

Damit ist auch bewiesen, daß der Algorithmus bei Erreichen von Schritt V. tatsächlich das Minimalpolynom $m_{A,b}$ ausgibt: Nach i. ist m ein Teiler von $m_{A,b}$, und wegen der Schleifenabbruchbedingung und iii. gilt $m(A)b = h = 0$, also muß $m = m_{A,b}$ sein. □

Für die Laufzeitanalyse des probabilistischen Algorithmus 2.4 folgen wir Abschnitt VI. in [Wie86]. Wiedemann bemerkt dort, daß der Algorithmus letztlich das kleinste gemeinsame Vielfache mehrerer Minimalpolynome $m_{u,A,b}$ für zufällig und unabhängig gewählte $u \in \mathbb{F}^n$ berechnet. Er gibt weiterhin an, daß die Wahrscheinlichkeit P_k dafür, daß nach k Versuchen das kgV gleich dem Minimalpolynom $m_{A,b}$ ist,

$$P_k = \prod_i (1 - q^{-kd_i})$$

beträgt. Das Produkt auf der rechten Seite geht dabei über alle verschiedenen irreduziblen Faktoren g_i von $m_{A,b}$, und d_i bezeichnet den Grad von g_i. Eine von den irreduziblen Faktoren unabhängige Abschätzung für diese Wahrscheinlichkeit ist ebenfalls in [Wie86] zu finden:

$$P_k \geq 1 - \log\left(\frac{q^{k-1}}{q^{k-1}-1}\right) \quad \text{für } k \geq 2$$

Vermöge der bekannten Ungleichung $\log x \leq x - 1$ kann der logarithmische Term durch

$$\begin{aligned} \log\left(\frac{q^{k-1}}{q^{k-1}-1}\right) &\leq \frac{q^{k-1}}{q^{k-1}-1} - 1 = \frac{1}{q^{k-1}-1} \\ &\leq \frac{1}{\frac{1}{2}q^{k-1}} = 2q^{1-k} \quad \text{für } k \geq 2 \end{aligned}$$

nach oben abgeschätzt werden.

Bezeichne $Q_k := P_k - P_{k-1}$ die Wahrscheinlichkeit dafür, daß *exakt* k Versuche nötig sind, um das Minimalpolynom zu erreichen. Der Erwartungswert $\mathcal{E}(N)$ für die Anzahl N benötigter Iterationen von Algorithmus 2.4 ergibt sich damit als

$$
\begin{aligned}
\mathcal{E}(N) &= \sum_{k\geq 1} kQ_k = \begin{array}{lllll} Q_1 & +Q_2 & +Q_3 & +Q_4 & +\dots \\ & +Q_2 & +Q_3 & +Q_4 & +\dots \\ & & +Q_3 & +Q_4 & +\dots \\ & & & +Q_4 & +\dots \\ & & & & \ddots \end{array} \\
&= \sum_{k\geq 0}(1-P_k) \\
&\leq 2 - P_0 - P_1 + \sum_{k\geq 2} \log\left(\frac{q^{k-1}}{q^{k-1}-1}\right) \\
&\leq 2 - P_0 - P_1 + \sum_{k\geq 2} 2q^{1-k} = 2 - P_0 - P_1 + 2\sum_{k\geq 1} q^{-k} \\
&= 2 - P_0 - P_1 + 2\frac{q^{-1}}{1-q^{-1}} = 2 - P_0 - P_1 + 2\frac{1}{q-1} \\
&\leq 4
\end{aligned}
$$

Die Kosten für die Schritte *einer* Iteration im einzelnen:

- Die Berechnung der a_i in Schritt III. erfordert $\mathcal{O}(n)$ Black-Box-Anwendungen von A auf einen Vektor und $\mathcal{O}(n)$ Skalarprodukte von $\mathbb{F}$-Vektoren der Dimension n, also insgesamt $\mathcal{O}(n(\mathrm{BB}(n)+n))$ Operationen in $\mathbb{F}$.
- Die Kosten für die Bestimmung des Minimalpolynoms in Schritt IV. betragen $\mathcal{O}(n^2)$ Körperoperationen, ebenso wie für die Aktualisierung von m.

  Die Berechnung von $m_1(A)h$ in Schritt IV. erfordert mit einem Horner-ähnlichen Verfahren $\mathcal{O}(\deg m_1)$ Black-Box-Anwendungen von A und ebenfalls $\mathcal{O}(\deg m_1)$ Additionen von n-dimensionalen $\mathbb{F}$-Vektoren. Die kumulierten Kosten dafür in *allen* Iterationen betragen also $\mathcal{O}(n(\mathrm{BB}(n)+n))$ Operationen in $\mathbb{F}$.

Damit hat Algorithmus 2.4 eine erwartete Laufzeit von $\mathcal{O}(n(\mathrm{BB}(n)+n))$ Körperoperationen in $\mathbb{F}$. Falls $\mathrm{BB}(n)$ mindestens linear in n wächst, was im allgemeinen der Fall ist, sind das $\mathcal{O}(n \cdot \mathrm{BB}(n))$ Operationen.

Bemerkung 2.5

1. Der Speicherplatzverbrauch von Algorithmus 2.4 beträgt lediglich $\mathcal{O}(n)$ (im Vergleich zu $\mathcal{O}(n^2)$ bei Speicherung der kompletten Matrix A): In Schritt III. ist für jedes i der Vektor $A^i h$ zu berechnen, der jedoch nach Berechnung von $u^{\mathbf{T}} \cdot A^i h$ und $A^{i+1}h = A(A^i h)$ nicht weiter gespeichert werden muß. Ähnliches gilt für die Berechnung von $m_1(A)h$ in Schritt IV.

2. Werden in Schritt III. alle Vektoren $A^i h$ $(0 \leq i \leq \text{maxdegree})$ gespeichert, so entfällt deren erneute Berechnung in Schritt IV. bei der Auswertung von $m_1(A)h$, die damit nur noch $\mathcal{O}(\deg m_1 \cdot n)$ Körperoperationen benötigt. Die asymptotische Laufzeit ändert sich dadurch nicht, der Speicherbedarf erhöht sich allerdings auf $\mathcal{O}(n^2)$.

3. Das Minimalpolynom m_A von A kann nach einem ähnlichen Schema berechnet werden wie $m_{A,b}$: Bestimme $m_{A,b}$ mit Algorithmus 2.4 für *mehrere*, unabhängig und zufällig gewählte $b \in \mathbb{F}^n$. Dann zeigt ein analoges Argument wie oben, daß das kleinste gemeinsame Vielfache aller $m_{A,b}$ mit hoher Wahrscheinlichkeit gleich m_A ist. Einzelheiten dazu finden sich in [Wie86].

2.2 Der Algorithmus von Kaltofen & Lobo

Wiedemanns Algorithmus, wie im vorigen Abschnitt vorgestellt, ist zunächst nur auf invertierbare Matrizen anwendbar. In [Wie86] wird zwar auch eine Variante von Algorithmus 2.3 für singuläre Matrizen vorgestellt, die aber nur für die „dünne" Repräsentation der Matrix und nicht allgemein für „black box"-Repräsentationen funktioniert.

Kaltofen und Saunders geben in [KS91] ein Las-Vegas-Verfahren an, welches die Bestimmung einer Lösung von $Ax = b$ für eine singuläre „black box"-Matrix[†] A mit algebraischen Methoden auf ein nichtsinguläres Gleichungssystem reduziert, das mit Wiedemanns Algorithmus gelöst werden kann. Die Methode kann dazu verwendet werden, ein zufälliges Element des Nullraums von A zu berechnen, wobei alle Elemente gleich wahrscheinlich sind. Sie ist daher insbesondere für „black box"-Repräsentationen der Matrizen $\overline{B}$ und N aus dem Berlekamp- und dem Niederreiter-Algorithmus geeignet.

Wir stellen nun ein Verfahren nach Kaltofen und Lobo [KL94] vor, welches ebenfalls für eine singuläre „black box"-Matrix A ein zufälliges Element des Nullraums bestimmt. Allerdings sind die möglichen Ergebnisse nicht gleichverteilt. Die Methode hat aber gegenüber der von Kaltofen und Saunders den Vorteil, daß keine algebraischen Manipulationen der Matrix A notwendig sind. Außerdem entfällt die in [KS91] erforderliche Bestimmung des Rangs von A.

Im weiteren ist $\mathbb{F} = \mathbb{F}_q$ ein endlicher Körper, $n \in \mathbb{N}$ und $A \in \mathbb{F}^{n \times n}$ eine als „black box" gegebene, singuläre Matrix. Zu lösen ist das folgende

Problem: Berechne ein beliebiges Element $h \neq 0$ des Nullraums von A, d.h. mit $Ah = 0$.

[†] Dieser Begriff wird in [KS91] geprägt

Algorithmus 2.6 (Kaltofen & Lobo)

{ *input: Dimension* $n > 0$, *„black box"-Algorithmus für die Anwendung von* $A \in \mathbb{F}^{n\times n}$, *A singulär, auf einen Vektor* $\in \mathbb{F}^n$ }

{ *output: Ein Vektor* $h \in \mathbb{F}^n \backslash \{0\}$ *mit* $Ah = 0$ }

I. *Setze* $d := 0$.

II. *Wiederhole die Schritte III. und IV. so lange, bis* $d > 0$*:*

III. *Wähle zufällig* $b \in \mathbb{F}^n$ *und berechne* $m := m_{A,b}$ *mit Algorithmus 2.4.*

IV. *Schreibe* $m = X^d \cdot m_1$ *mit* $d \in \mathbb{N}$ *und* $m_1(0) \neq 0$.

V. *Setze* $h := A^{d-1} \cdot m_1(A)b$.

VI. *output* h.

Korrektheitsbeweis: Die Schleife II. wird offenbar mindestens einmal durchlaufen; nach deren Abbruch ist $d \geq 1$ in Schritt V. Nach Konstruktion gilt $Ah = A^d \cdot m_1(A)b = m(A)b = 0$. Andererseits ist $h = A^{d-1} \cdot m_1(A)b = \frac{m}{X}(A)b \neq 0$ nach Definition des Minimalpolynoms. □

Die Kosten für die einzelnen Schritte betragen, gemessen in $\mathbb{F}$-Körperoperationen:

Schritt III.	$\mathcal{O}(n \cdot \mathrm{BB}(n))$ (erwartete Laufzeit)
Schritt V.	$\mathcal{O}(n \cdot \mathrm{BB}(n))$

Die nachfolgende Analyse wird zeigen, daß die erwartete Anzahl von Durchläufen der Schleife II. konstant ist. Insgesamt hat der Algorithmus damit eine erwartete Laufzeit von $\mathcal{O}(n(\mathrm{BB}(n) + n))$ Körperoperationen in $\mathbb{F}$, das sind $\mathcal{O}(n \cdot \mathrm{BB}(n))$ bei mindestens linearem Wachstum von $\mathrm{BB}(n)$.

Um zu veranschaulichen, warum der Algorithmus von Kaltofen und Lobo funktioniert, und um ihn stochastisch analysieren zu können, betrachten wir die Eigenwerte des Endomorphismus $\alpha : x \mapsto Ax$.

Seien $\overline{\mathbb{F}}$ ein algebraischer Abschluß von $\mathbb{F}$ und $\lambda_1, \ldots, \lambda_t \in \overline{\mathbb{F}}$ die verschiedenen Eigenwerte von α. Der Vektorraum $V := \overline{\mathbb{F}}^n \supseteq \mathbb{F}^n$ läßt sich wie folgt zerlegen:

$$V = \bigoplus_{1 \leq i \leq t} W_i \tag{2.1}$$

W_i ist dabei der *Hauptraum* von V zum Eigenwert λ_i mit den Eigenschaften

1. $\alpha W_i \subseteq W_i$
2. W_i enthält den Eigenraum $E_i := \ker(\alpha - \lambda_i \mathrm{id}_V)$ zum Eigenwert λ_i. Die Dimension γ_i von E_i ist die *geometrische Vielfachheit* von λ_i.
3. Es gibt eine natürliche Zahl $l \geq 1$, so daß $(\alpha - \lambda_i \mathrm{id}_V)^l w_i = 0$ für alle $w_i \in W_i$ ist. Die *kleinste* solche Zahl nennen wir δ_i. Sie heißt *maximale Vielfachheit* von λ_i.
4. $\dim_{\mathbb{F}} W_i$ ist die *algebraische Vielfachheit* des Eigenwerts λ_i.

Das Minimalpolynom von α — eingeschränkt auf W_i — ist $(X - \lambda_i)^{\delta_i}$ nach Eigenschaft 3. Das Minimalpolynom m_A von α auf ganz V ergibt sich damit als $\prod_{1 \leq i \leq t} (X - \lambda_i)^{\delta_i}$.

Für einen Vektor $w_i \in W_i$ definieren wir die *Höhe* von w_i bzgl. λ_i als $H_i(w_i) := \min\{k \geq 0;\ (\alpha - \lambda_i \mathrm{id}_V)^k w_i = 0\}$. Es gilt:

- $H_i(0) = 0$
- $H_i(w_i) = 1 \Leftrightarrow w_i$ Eigenvektor zum Eigenwert λ_i
- $H_i(w_i) \leq \delta_i$ (Eigenschaft 3.)

Das Minimalpolynom von $(w_i, \alpha w_i, \alpha^2 w_i, \alpha^3 w_i, \ldots)$ ist also $(X - \lambda_i)^{H_i(w_i)}$.

Wir können jeden Vektor $b \in V$ gemäß der Zerlegung (2.1) eindeutig als Linearkombination $b = \sum_{1 \leq i \leq t} w_i$ mit $w_i \in W_i$ schreiben. Für einen Teiler $m = \prod_{1 \leq i \leq t} (X - \lambda_i)^{e_i}$ $(e_i \in \mathbb{N})$ von m_A gilt:

$$
\begin{aligned}
m(\alpha)b = 0 \quad &\Leftrightarrow \quad \forall i : m(\alpha) w_i = 0 \quad \text{(wegen (2.1) und Eigenschaft 1.)} \\
&\Leftrightarrow \quad \forall i : (X - \lambda_i)^{H_i(w_i)} \text{ teilt } m \\
&\Leftrightarrow \quad \forall i : H_i(b) \leq e_i
\end{aligned}
$$

mit $H_i(b) := H_i(w_i)$. Insbesondere ist $m_{A,b} = \prod_{1 \leq i \leq t} (X - \lambda_i)^{H_i(b)}$.

Daraus ergeben sich folgende Konsequenzen:

1. Die Matrix A (und damit der Endomorphismus α) ist nach Voraussetzung singulär, hat also 0 als Eigenwert. Wir nehmen daher o.E. $\lambda_1 = 0$ an. Die Schleife II. von Algorithmus 2.6 wird erfolgreich abgebrochen, wenn das Minimalpolynom $m_{A,b}$ für den in Schritt III. zufällig gewählten Vektor $b \in \mathbb{F}^n$ durch X teilbar ist, d.h. $H_1(b) > 0$, oder äquivalent, $w_1 \neq 0$.

Selbst im ungünstigsten Fall, nämlich wenn die algebraische Vielfachheit des Eigenwerts 0 nur $\dim_{\overline{\mathbb{F}}} W_1 = 1$ beträgt, ist die Wahrscheinlichkeit für ein „erfolgreiches" b (d.h. mit nicht verschwindendem $w_1 \in W_1 \cap \mathbb{F}^n$) mindestens $1 - \frac{1}{q} \geq \frac{1}{2}$. Damit ist die erwartete Anzahl Iterationen der Schleife II. höchstens 2.

2. Sei nun $b \in \mathbb{F}^n$ ein „erfolgreicher" Vektor,

$$m := m_{A,b} = X^d m_1 = (X - \lambda_1)^d m_1$$

mit $d = H_1(b) \geq 1$ wie in Algorithmus 2.6 und $u := m_1(A)b = m_1(\alpha)b$. Dann gilt nach Konstruktion $(\alpha - \lambda_1 \mathrm{id}_V)^d u = \alpha^d u = m(\alpha)b = 0$, d.h. $u \in W_1 \cap \mathbb{F}^n$. Die Anwendung des X-freien Anteils $m_1(\alpha)$ auf den Vektor b annulliert also alle Komponenten w_i von b, die *nicht* zum Eigenwert $\lambda_1 = 0$ gehören (nämlich die für $i \geq 2$):

$$\begin{array}{ccccccccccc} b & = & w_1 & + & w_2 & + & \ldots & + & w_t & & \\ & \overset{m_1(\alpha)}{\longmapsto} & w_1' & + & 0 & + & \ldots & + & 0 & = & u \end{array} \tag{2.2}$$

(dabei ist $w_1' = m_1(\alpha)w_1 = u \in W_1 \cap \mathbb{F}^n$).

Alle möglichen $u \in (W_1 \cap \mathbb{F}^n) \backslash \{0\}$ treten mit gleicher Wahrscheinlichkeit auf, wenn $b \in \mathbb{F}^n$ mit $w_1 \neq 0$ zufällig und uniform gewählt wird, denn: Wegen $X \nmid m_1$ gibt es $s, t \in \mathbb{F}[X]$ mit $s \cdot X^{\delta_1} + t \cdot m_1 = 1$. Daraus folgt $t(\alpha)m_1(\alpha)w_1 = s(\alpha)\underbrace{\alpha^{\delta_1} w_1}_{=0} + t(\alpha)m_1(\alpha)w_1 = w_1$ für alle $w_1 \in W_1$. Die auf W_1 eingeschränkte Abbildung $m_1(\alpha) : W_1 \longrightarrow W_1$ ist also invertierbar und damit bijektiv.

Um die Verteilung der Ergebnisvektoren h zu bestimmen, kehren wir wieder in den Vektorraum $\mathbb{F}^n$ über dem ursprünglichen Körper zurück. Folgende Abkürzungen werden verwendet:

- $W := W_1 \cap \mathbb{F}^n$, der Hauptraum zum Eigenwert 0 über $\mathbb{F}$,
- $E := E_1 \cap \mathbb{F}^n$, der Eigenraum zum Eigenwert 0 (d.h. der Nullraum von A) über $\mathbb{F}$,
- $\gamma := \dim_{\mathbb{F}} E = \gamma_1 = \dim_{\overline{\mathbb{F}}} E_1$ dessen Dimension, die geometrische Vielfachheit,
- $\delta := \delta_1 = \min\{k \geq 0;\ \alpha^k u = 0\ \forall u \in W\}$, die maximale Vielfachheit,
- $H(u) := H_1(u)$, die Höhe von $u \in W$ bezüglich des Eigenwerts 0.

Alle Dimensionsangaben beziehen sich, falls nicht explizit angegeben, auf den Grundkörper $\mathbb{F}$.

Wir betrachten folgendes Zufallsexperiment:

> Wähle zufällig und uniform einen Vektor $u \in W \backslash \{0\}$ und berechne $h := \alpha^{d-1}u$ für $d := H(u)$.

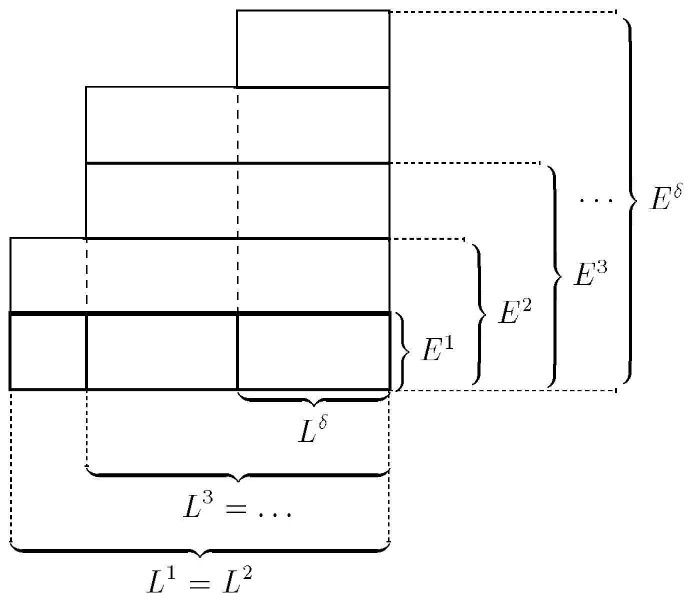
Die gesuchte Wahrscheinlichkeit $P(h)$ dafür, daß der bei diesem Experiment berechnete Vektor gleich einem vorgegebenen Vektor $h \in E \backslash \{0\}$ im Nullraum von α ist, hängt von der Struktur des Hauptraums $W = \ker \alpha^\delta$ ab. Wir setzen $E^k := \ker \alpha^k$ für $k \geq 0$. Damit gilt $H(u) = k \Leftrightarrow u \in E^k \backslash E^{k-1}$. Die E^k bilden eine aufsteigende Kette von Unterräumen in W:

$$E^0 = \{0\} \subseteq E^1 = E \subseteq \ldots \subseteq E^k \subseteq \ldots \subseteq E^\delta = E^{\delta+1} = \ldots = W$$

Jede Anwendung von α verringert die Höhe der Vektoren in dieser Kette um 1, „verschiebt“ also die Kette eins nach links, denn $\alpha E^k = \alpha \ker \alpha^k \subseteq \ker \alpha^{k-1} = E^{k-1}$ für $k \geq 1$. Das Bild von E^k in E unter der Abbildung α^{k-1} $(k \geq 1)$ bezeichnen wir mit L^k. Nach der Dimensionsformel gilt $\dim L^k = \dim E^k - \dim E^{k-1}$. Die L_k bilden eine absteigende Kette von Unterräumen in E:

$$L^1 = E \supseteq L^2 \supseteq \ldots \supseteq L^k \supseteq \ldots \supseteq L^\delta \supseteq L^{\delta+1} = L^{\delta+2} = \ldots = \{0\}$$

Bildlich sieht die Situation so aus:



Die Fläche in diesem Balkendiagramm symbolisiert die Dimension der jeweiligen Unterräume. Die Gesamtfläche der Balken ist also die algebraische Vielfachheit des Eigenwerts 0. Die vertikale Position gibt die Höhe der Vektoren an, die Breite des untersten Balkens die geometrische Vielfachheit γ. Eine Anwendung von α „kappt“ die unterste Schicht des Balkendiagramms, die darüberliegenden Schichten „sinken“ um eins nach unten.
Im Beispiel ist die algebraische Vielfachheit $\dim W = 2+4+4+5+5 = 20$, die geometrische Vielfachheit (die Dimension des Nullraums von α) $\gamma = 5$ und die maximale Vielfachheit (der Grad des Minimalpolynoms von α auf W) $\delta = 5$. Ferner sind $\dim E^1 = \gamma = 5$, $\dim E^2 = 10$, $\dim E^3 = 14$, $\dim E^4 = 18$ und $\dim E^5 = \dim W = 20$.

Sei nun $h \in L^k \backslash L^{k+1} \subseteq E\backslash\{0\}$ für $1 \leq k \leq \delta$ das Ergebnis des Zufallsexperiments für ein zufällig gewähltes $u \in W\backslash\{0\}$, d.h. $\alpha^{d-1}u = h$ für $d := H(u) \geq 1$. Dann muß — wegen $h \notin L^{k+1}$ — $d \leq k$ sein. Für ein festes d mit $1 \leq d \leq k$ hat h genau die Vektoren aus $h + E^{d-1}$ als Urbilder unter der Abbildung α^{d-1}. Insgesamt gibt es

$$\sum_{1 \leq d \leq k} \#E^{d-1} = \sum_{0 \leq d < k} \#E^d = \sum_{0 \leq d < k} q^{\dim E^d}$$

Möglichkeiten, h als Ergebnis des Zufallsexperiments zu erhalten. Für die Wahrscheinlichkeit $P(h)$ gilt daher

$$P(h) = \frac{1}{q^{\dim W} - 1} \sum_{0 \leq d < k} q^{\dim E^d} \quad \text{falls } h \in L^k \backslash L^{k+1} \quad (1 \leq k \leq \delta) \tag{2.3}$$

Die Verteilung im Beispiel:

$h \in$	Anzahl	$P(h)$	Größenordnung
$L^2 \backslash L^3$	$q^5 - q^4$	$\dfrac{1+q^5}{q^{20}-1}$	$\mathcal{O}(q^{-15})$
$L^4 \backslash L^5$	$q^4 - q^2$	$\dfrac{1+q^5+q^{10}+q^{14}}{q^{20}-1}$	$\mathcal{O}(q^{-6})$
$L^5 \backslash \{0\}$	$q^2 - 1$	$\dfrac{1+q^5+q^{10}+q^{14}+q^{18}}{q^{20}-1}$	$\mathcal{O}(q^{-2})$

Je „weiter rechts“ h in der L^k-Hierarchie liegt (d.h. je größer $\max\{k;\ 1 \leq k \leq \delta \text{ und } h \in L^k\}$ ist), desto häufiger tritt h als Ergebnis des Zufallsexperiments (und damit von Algorithmus 2.6) auf! Im Beispiel etwa ist es bei großem q sehr unwahrscheinlich (die Wahrscheinlichkeit dafür ist $\mathcal{O}(q^{-2})$), daß h nicht in $L^5\backslash\{0\}$ liegt, obwohl $L^5\backslash\{0\}$ nur q^2-1 der q^5-1 Elemente von $E\backslash\{0\}$ enthält.

2.3 Faktorisieren mit „Black Box“-Matrizen

Kaltofen und Lobo wenden Algorithmus 2.6 in [KL94] nur auf die Matrix B aus Berlekamps Algorithmus an. Sie zeigen für diesen Fall, daß die Verteilung der Ergebnisvektoren h „hinreichend zufällig“ für einen „split“ des zu faktorisierenden Polynoms ist. Wir geben eine Variante des Verfahrens für die Matrix $\overline{B}$ (vgl. Bem. 1.7) an, werden die Analyse aus [KL94] in dieser Situation nachvollziehen und auch untersuchen, wie sich der Algorithmus für die Niederreiter-Matrix N verhält.

Seien $\mathbb{F} = \mathbb{F}_p \subseteq \mathbb{F}_q$ endliche Körper der Charakteristik p, $q = p^r$ und $f \in \mathbb{F}_q[X]$ ein zu faktorisierendes Polynom vom Grad n, normiert und quadratfrei, mit kanonischer Faktorisierung $f = g_1 \cdot \ldots \cdot g_t$ ($g_i \in \mathbb{F}_q[X]$ irreduzibel und paarweise verschieden). Ferner bezeichne

$$R := \mathbb{F}_q[X]/(f) \cong \mathbb{F}_q[X]/(g_1) \times \ldots \times \mathbb{F}_q[X]/(g_t) = \mathbb{F}_{q^{d_1}} \times \ldots \times \mathbb{F}_{q^{d_t}}$$

wobei $d_i := \deg g_i$ $(1 \leq i \leq t)$. R ist eine nr-dimensionale $\mathbb{F}_p$-Algebra. Den p-Frobenius-Endomorphismus $h \mapsto h^p$ von R notieren wir mit $\overline{\varphi}$.

Die Matrix $\overline{B}$ ergibt sich aus der Darstellung des $\mathbb{F}_p$-Endomorphismus

$$\overline{\beta} := \overline{\varphi} - \mathrm{id} : \begin{cases} R & \longrightarrow & R \\ h & \mapsto & h^p - h \end{cases}$$

bezüglich einer geeigneten $\mathbb{F}_p$-Basis von R.

Algorithmus 2.7 (Kaltofen & Lobo)

{ *input:* $f \in \mathbb{F}_q[X]$ *normiert, quadratfrei,* $\deg f = n$, *Schranke* $m \in \mathbb{N}$ }

{ *output: Mit einer gewissen Wahrscheinlichkeit* $g_1, \ldots, g_t$, *die irreduziblen Faktoren von* f }

I. *Setze factorbase* $:= \{f\}$.

II. *Wiederhole Schritt III. bis zu* m *mal:*

III. *Berechne mit Algorithmus 2.6, angewandt auf eine „black box“-Repräsentation der Matrix* $\overline{B}$, *ein Element* h *der absoluten Berlekamp-Algebra* $\overline{\mathcal{B}}$. *Falls* $h \notin \mathbb{F}_p$, *brich die Schleife II. ab und fahre mit Schritt V. fort.*

IV. *Weiter mit Schritt VIII.*

{ *Falls* $p > 2$*:* }

V. *Wiederhole Schritt VI. für ein zufällig gewähltes* $a \in \mathbb{F}_p$ *so lange, bis* $\#factorbase > 1$*:*

VI. (split) *Berechne* $f_1 := ggT(h-a, f)$, $f_2 := ggT((h-a)^{\frac{p-1}{2}} - 1 \bmod f, f)$ *und* $f_3 := \frac{f}{f_1 f_2}$ *und setze* $factorbase := factorbase \cup (\{f_1, f_2, f_3\} \backslash \{1\})$.

{ *Falls* $p = 2$*:* }

VI. (split) *Berechne* $f_1 := ggT(h, f)$ *und* $f_2 := \frac{f}{f_1}$ *und setze* $factorbase := \{f_1, f_2\}$.

VII. *Rufe Algorithmus 2.7 rekursiv für alle Faktoren* $\in$ *factorbase auf und sammle die zurückgelieferten Ergebnisse in factorbase.*

VIII. *output factorbase.*

Bei diesem Algorithmus handelt es sich um ein „Monte Carlo“-Verfahren, d.h. das Ergebnis ist nur mit einer gewissen, in diesem Fall von m abhängigen Wahrscheinlichkeit korrekt. Offenbar wird stets eine Faktorisierung von f berechnet, ob diese jedoch vollständig ist, d.h. die einzelnen Faktoren irreduzibel sind, kann nicht garantiert werden. Die stochastische Analyse wird jedoch zeigen, daß im reduziblen Fall mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ in Schritt III. ein h berechnet wird, welches schließlich in Schritt VI. zu einem „split“ von f führt.

Rechenzeitanalyse:

- Eine „black box“-Anwendung der Matrix $\overline{B}$ ist einfach zu berechnen, denn für $h \in R$ ist $\overline{B}h = \overline{\beta}h = h^p - h$. Die Kosten dafür betragen $\mathcal{O}(\log p)$ Multiplikationen in R, das sind $\mathcal{O}(\log p \cdot M(n))$ Operationen in $\mathbb{F}_q$ oder $\mathcal{O}(\log p \cdot M(n) \cdot M(r)) =: \mathrm{BB}(n,r)$ Operationen in $\mathbb{F}_p$.
- Die (erwarteten) Kosten für Schritt III. betragen $\mathcal{O}(nr \cdot \mathrm{BB}(n,r))$ Operationen in $\mathbb{F}_p$. Die Anzahl Durchläufe der Schleife II. beträgt $\mathcal{O}(m)$.
- Die erwartete Anzahl Durchläufe der Schleife V. im Fall $p > 2$ ist konstant, denn: Für $h = (h_1, \ldots, h_t) \in \overline{\mathcal{B}} \backslash \mathbb{F}_p$ gibt es Indizes $1 \leq i < j \leq t$ mit $h_i \neq h_j$. Damit ist nach [Rab80] die Wahrscheinlichkeit dafür, daß $(h_i - a)^{\frac{p-1}{2}} \neq (h_j - a)^{\frac{p-1}{2}}$ für ein zufällig gewähltes $a \in \mathbb{F}_p$ ist, gleich $\frac{1}{2} - \frac{1}{2p} \geq \frac{1}{3}$. Die Faktoren g_i und g_j von f werden also mit einer Wahrscheinlichkeit von mindestens $\frac{1}{3}$ in Schritt VI. getrennt, und der Erwartungswert für die Anzahl Schleifendurchläufe beträgt höchstens 3.

Im Fall $p = 2$ liefert *jedes* $h = (h_1, \ldots, h_t) \in \overline{\mathcal{B}} \backslash \{0, 1\}$ einen nichttrivialen ggT in Schritt VI., da dann Indizes $1 \leq i \neq j \leq t$ existieren mit $h_i = 0$ und $h_j = 1$, d.h. $g_i | h$ und $g_j \nmid h$.

Die Kosten für Schritt VI. werden von den Kosten für Schritt III. dominiert.

- Die erwartete Gesamtlaufzeit für Algorithmus 2.7 ergibt sich als Summe der Kosten über alle rekursiven Aufrufe. Da die Summe der Grade der auf einer Rekursionsebene behandelten Faktoren von f höchstens n beträgt, sind die Kosten für die Berechnungen auf jeder Rekursionsebene höchstens so groß wie die im allerersten Schritt.

  Für die Bestimmung der erwarteten Rekursionstiefe τ verweisen wir auf [KL94]. Damit haben wir insgesamt eine erwartete Laufzeit von $\mathcal{O}(\tau \cdot m \cdot nr \cdot \mathrm{BB}(n, r)) = \mathcal{O}(\tau \cdot m \cdot \log p \cdot nM(n) \cdot rM(r))$ Körperoperationen in $\mathbb{F}_p$.

Der Speicherbedarf beträgt $\mathcal{O}(nr)$ Plätze für Elemente von $\mathbb{F}_p$. Nun zur Analyse der Erfolgswahrscheinlichkeit des Algorithmus. Im Gegensatz zu allen bis dato vorgestellten Faktorisierungsverfahren ist hier die Anzahl t der Faktoren nicht bekannt, also kann kein definitives Abbruchkriterium angegeben werden. Statt dessen geht der Algorithmus so vor, daß nach m erfolglosen Versuchen, ein $h \in \overline{\mathcal{B}} \backslash \mathbb{F}_p$ zu finden, angenommen wird, daß das Polynom f irreduzibel ist. Die Wahrscheinlichkeit dafür, daß Algorithmus 2.7 die komplette Faktorisierung berechnet, hängt von m und der Wahrscheinlichkeit ab, daß das in Schritt III. berechnete h zu einem „split“ führt, d.h. nicht in $\mathbb{F}_p$ liegt. Letztere wollen wir nun bestimmen.

Gemäß der Analyse in Abschnitt 2.2 müssen wir dazu den Hauptraum W von R zum Eigenwert 0 von $\overline{\beta}$ untersuchen. Wegen $\overline{\beta} = \overline{\varphi} - \mathrm{id}$ ist W gleichzeitig der Hauptraum und $\overline{\mathcal{B}}$ der Eigenraum von R zum Eigenwert $\lambda = 1$ von $\overline{\varphi}$. Für die Minimalpolynome von $\overline{\beta}$ und $\overline{\varphi}$ über $\mathbb{F}_p$ gilt $m_{\overline{\beta}}(X) = m_{\overline{\varphi}}(X + 1)$. Es genügt also, das Minimalpolynom des Frobenius-Endomorphismus $\overline{\varphi}$ und den zugehörigen Hauptraum W zum Eigenwert 1 zu betrachten.

Das Minimalpolynom von $\overline{\varphi}$ auf einer Komponente $\mathbb{F}_{q^{d_i}} = \mathbb{F}_{p^{rd_i}}$ von R ist $X^{rd_i} - 1$, denn: Offenbar wird $\overline{\varphi}$ von diesem Polynom annulliert, und nach Dedekinds Lemma ([LN83] Kap. 2 Lemma 2.33) von keinem Polynom kleineren Grades. Aus Gradgründen ist $X^{rd_i} - 1$ auch das charakteristische Polynom von $\overline{\varphi}$ auf der Komponente $\mathbb{F}_{q^{d_i}}$. Es ist quadratfrei, falls $p \nmid (rd_i)$. Andernfalls tritt jeder irreduzible Faktor mit Vielfachheit p^{e_i} auf, wobei $e_i := \max\{e \in \mathbb{N};\ p^e | (rd_i)\}$. Insbesondere hat der zum Eigenwert $\lambda = 1$ gehörende Linearfaktor $X - 1$ die Vielfachheit p^{e_i}, welches gleichzeitig die algebraische und die maximale Vielfachheit von λ auf der Komponente $\mathbb{F}_{q^{d_i}}$ ist.

Das Minimalpolynom $m_{\overline{\varphi}}$ des p-Frobenius-Endomorphismus von R ist

$$m_{\overline{\varphi}} = \underset{1 \leq i \leq t}{\mathrm{kgV}} \left(X^{rd_i} - 1 \right)$$

und der Faktor $X-1$ hat darin die Vielfachheit $\delta := \max_{1 \leq i \leq t} p^{e_i}$. Der Hauptraum $W := \ker(\overline{\varphi} - \mathrm{id})^{\delta} = \ker \overline{\beta}^{\delta}$ von R zum Eigenwert 1 von $\overline{\varphi}$ ist das direkte Produkt der jeweiligen Haupträume der einzelnen Komponenten.

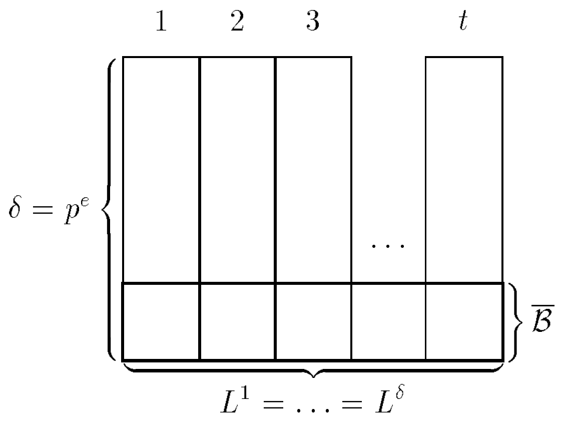
Sei nun $t \geq 2$, $u = (u_1, \ldots, u_t)$ ein zufällig (und uniform) gewähltes, von 0 verschiedenes Element von W und $h = (h_1, \ldots, h_t) := (\overline{\varphi} - \mathrm{id})^{d-1} u \in \overline{\mathcal{B}} \backslash \{0\}$ für $d := H(u)$, die Höhe von u bzgl. des Eigenwerts 1 von $\overline{\varphi}$. Wir interessieren und für die Wahrscheinlichkeit, daß zwei Indizes $1 \leq i < j \leq t$ existieren mit

$$(\overline{\varphi} - \mathrm{id})^{d-1} u_i = h_i \neq h_j = (\overline{\varphi} - \mathrm{id})^{d-1} u_j$$

Dazu unterscheiden wir zwei Fälle:

1. $e_1 = e_2 = \ldots = e_t =: e$. Dann ist $L^{\delta} = \overline{\mathcal{B}}$, und alle $h \in \overline{\mathcal{B}} \backslash \{0\}$ treten in Schritt III. mit gleicher Wahrscheinlichkeit auf. Die Wahrscheinlichkeit dafür, daß h zu einem „split“ führt, d.h. daß nicht $h_1 = \ldots = h_t$ gilt, beträgt damit $1 - \frac{p-1}{p^t-1} \geq \frac{2}{3}$.

   Anschaulich haben wir folgende Situation:



   Jeder senkrechte Balken stellt den Hauptraum zum Eigenwert 1 von $\overline{\varphi}$ in einer Komponente $\mathbb{F}_{q^{d_i}}$ dar, seine Höhe dessen $\mathbb{F}_p$-Dimension $\delta = p^e$.

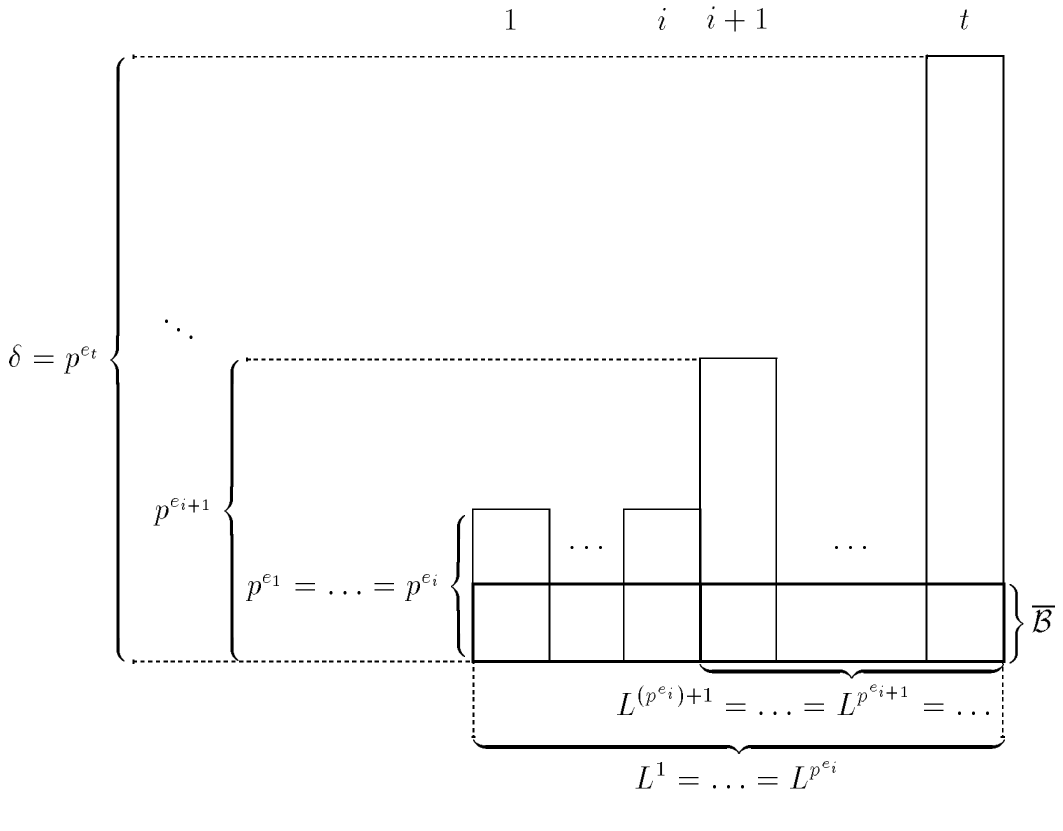
2. $\exists i \neq j : e_i < e_j$. Ohne Einschränkung nehmen wir an, daß die e_k der Größe nach aufsteigend geordnet sind, e_i minimal und $j = i+1$ ist, d.h.

$$e_1 = e_2 = \ldots = e_i < e_{i+1} \leq \ldots \leq e_t$$

Alle $h \in L^{p^{e_i}} \backslash L^{(p^{e_i})+1} = \overline{\mathcal{B}} \backslash L^{p^{e_{i+1}}}$ treten mit der gleichen Wahrscheinlichkeit auf. Ein solches h ist von der Form $h = (h_1, \ldots, h_t)$ mit $h_k \neq 0$ für $k \leq i$. Die bedingte Wahrscheinlichkeit dafür, daß so ein h zu einem „split“ führt, beträgt $1 - \frac{p-1}{(p-1)^i p^{t-i}} \geq \frac{1}{2}$.

Ist hingegen $h \in L^{p^{e_{i+1}}} \backslash \{0\}$, so führt h mit Sicherheit zu einem „split“, denn dann hat h die Form $h = (0, \ldots, 0, h_{i+1}, \ldots, h_t)$ mit $h_k \neq 0$ für mindestens ein $i < k \leq t$.

Bildlich:



In jedem Fall beträgt die Wahrscheinlichkeit für $h \notin \mathbb{F}_p$ in Schritt III. mindestens $\frac{1}{2}$. Die Wahrscheinlichkeit dafür, daß f reduzibel ist, aber wegen m-maligen Mißerfolgs in Schritt III. fälschlicherweise als irreduzibel angenommen wird, ist also höchstens $(\frac{1}{2})^m$. Dies gilt für jeden Faktor in der Menge factorbase, so daß die Fehlerwahrscheinlichkeit von Algorithmus 2.7 höchstens

$$\begin{aligned} 1-\left(1-2^{-m}\right)^n &\le 1-\left(1-\frac{2^{-k}}{n}\right)^n \quad \text{für } m=\lceil \log_2 n\rceil + k, \quad k\in\mathbb{N} \\ &\le 1-(1-2^{-k}) = 2^{-k} \end{aligned}$$

beträgt und damit beliebig klein gemacht werden kann, ohne die erwartete Laufzeit um mehr als einen logarithmischen Faktor zu erhöhen.

Bemerkung 2.8

1. In einer Implementation von Algorithmus 2.7 ist es sinnvoll, die Rekursion unterhalb einer geeigneten Gradschranke abzubrechen und auf Gauß-Elimination (d.h. Algorithmus 1.6) umzuschalten.

2. Mit wachsendem p steigt auch die Wahrscheinlichkeit dafür, daß das in Schritt III. produzierte $h = (h_1, \ldots, h_t) \in \overline{\mathcal{B}}$ mehr als zwei verschiedene Komponenten h_i hat. Vor dem rekursiven Aufruf sollten daher bei großem p mehrere „split“-Versuche in den Schritten V. und VI. gemacht und anschließend verfeinert werden, was f mit hoher Wahrscheinlichkeit in mehr als zwei Faktoren zerlegt.

3. Die Kosten für Algorithmus 2.7, angewandt auf die Matrix B statt $\overline{B}$, unterscheiden sich nur geringfügig (im Gegensatz zu den Kosten für die „absolute“ bzw. „relative“ Version von Berlekamps Algorithmus, vgl. Bem. 1.7). Eine Anwendung des relativen „black box“-Operators

$$\beta : \begin{cases} R & \longrightarrow \quad R \\ h & \longmapsto \quad h^q - h \end{cases}$$

erfordert $\mathcal{O}(\log q \cdot M(n)) = \mathcal{O}(r \log p \cdot M(n))$ Operationen in $\mathbb{F}_q$ bzw. $\mathcal{O}(r \log p \cdot M(n)M(r))$ Operationen in $\mathbb{F}_p$. Die zu β gehörige Matrix B hat Dimension n über $\mathbb{F}_q$, daher sind für eine Minimalpolynom-Berechnung $\mathcal{O}(n)$ „black box“-Anwendungen notwendig.

   Insgesamt ergibt das eine Laufzeit von $\mathcal{O}(\tau \cdot m \cdot \log p \cdot nM(n) \cdot rM(r))$ Operationen in $\mathbb{F}_p$ — dieselbe Größenordnung wie bei Algorithmus 2.7 —, wobei τ und m nur von der Anzahl t der Faktoren abhängen und nicht von der Körpergröße. Der Speicherbedarf für beide Versionen beträgt $\mathcal{O}(nr)$ Plätze für Elemente von $\mathbb{F}_p$.

Algorithmus 2.7 kann so abgewandelt werden, daß statt der Matrix $\overline{B}$ die Niederreiter-Matrix N verwendet wird. Dazu ist eine „black box“-Version der Matrix N nötig. In Schritt III. wird ein Element $h \in \mathcal{N}$ berechnet und muß auf $h \notin \mathbb{F}_p \cdot f'$ getestet werden. Falls $p > 2$, so ist in Schritt VI. $(h-a)^{\frac{p-1}{2}} - 1$ durch $(h - af')^{\frac{p-1}{2}} - (f')^{\frac{p-1}{2}}$ zu ersetzen (vgl. Lemma 1.14).

Für eine „black box“-Anwendung der Niederreiter-Matrix auf einen Vektor (ein Polynom) h ist der Ausdruck

$$\left(f^p \left(\frac{h}{f}\right)^{(p-1)} + h^p\right)(X^{\frac{1}{p}})$$

zu berechnen. In dem für eine Implementation interessanten Fall $p = 2$ vereinfacht sich das zu

$$\begin{aligned}\left(f^2 \left(\frac{h}{f}\right)' + h^2\right)(\sqrt{X}) &= (h'f + hf' + h^2)(\sqrt{X}) \\ &= ((hf)' + h^2)(\sqrt{X})\end{aligned}$$

Die Kosten dafür sind im wesentlichen die für eine Polynommultiplikation, das sind $\mathcal{O}(M(n))$ Operationen in $\mathbb{F}_q$ oder $\mathcal{O}(M(n)M(r))$ Operationen in $\mathbb{F}_2$. Sie liegen in derselben Größenordnung wie die Kosten einer „black box“-Anwendung der Matrix $\overline{B}$ für $p = 2$.

Leider ist die Erfolgswahrscheinlichkeit für diese Variante von Algorithmus 2.7 nicht so hoch wie im Berlekamp-Fall. Der Hauptraum W zum Eigenwert 0 von N „verteilt“ sich nicht über die einzelnen irreduziblen Faktoren von f. Dies hat zur Folge, daß die Wahrscheinlichkeit für einen „split“ deutlich unter $\frac{1}{2}$ sinken kann.

Beispiel: $p = 2$, $r = 10$, $\mathbb{F} = \mathbb{F}_2$, $\mathbb{F}_q = \mathbb{F}_{1024}$, gegeben als $\mathbb{F}_2[X]/(X^{10} + X^3 + 1)$ mit erzeugendem Element $\alpha := X + (X^{10} + X^3 + 1)$. Zu faktorisieren ist das Polynom

$$f = X^2 + (1 + \alpha^2 + \alpha^3 + \alpha^4 + \alpha^6 + \alpha^9)X + (\alpha + \alpha^2 + \alpha^5 + \alpha^6 + \alpha^8 + \alpha^9) \in \mathbb{F}_{1024}[X]$$

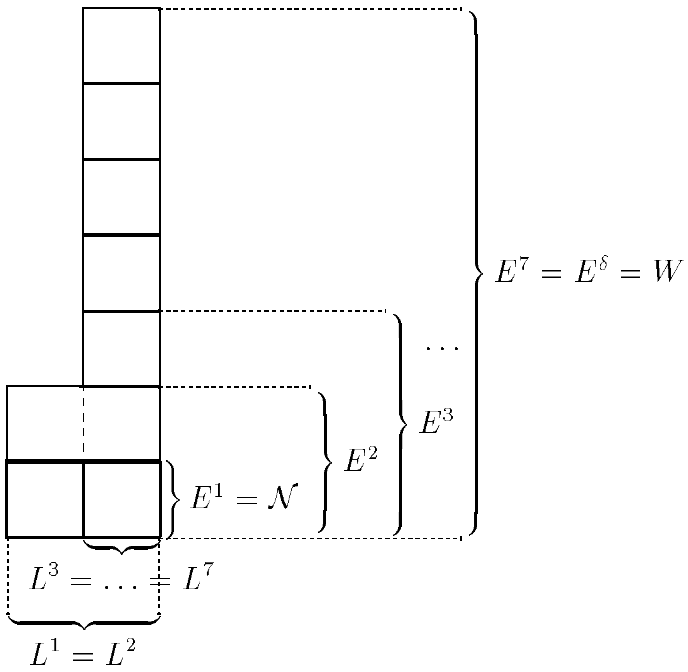
Es zerfällt in die zwei Linearfaktoren

$$g_1 := X + \alpha^8, \quad g_2 := X + (1 + \alpha^2 + \alpha^3 + \alpha^4 + \alpha^6 + \alpha^8 + \alpha^9)$$

Die Niederreiter-Matrix von f ist die Darstellung des $\mathbb{F}_2$-linearen Operators

$$\nu : \begin{cases} \mathbb{F}_{1024}[X]_2 & \longrightarrow & \mathbb{F}_{1024}[X]_2 \\ h(X) & \mapsto & ((hf)' + h^2)(\sqrt{X}) \end{cases}$$

bzgl. einer geeigneten $\mathbb{F}_2$-Basis von $\mathbb{F}_{1024}[X]_2$. Der Hauptraum W von $\mathbb{F}_{1024}[X]_2$ zum Eigenwert 0 von ν hat die folgende Struktur:



Die Daten sind $\delta = 7$, $\gamma = \dim \mathcal{N} = \dim E^1 = 2$, $\dim E^2 = 4$, $\dim E^3 = 5$, $\dim E^4 = 6$, $\dim E^5 = 7$, $\dim E^6 = 8$, $\dim E^7 = \dim E^\delta = \dim W = 9$. Der Nullraum $\mathcal{N}$ besteht aus den 4 Elementen $\{0, f', g_1, g_2 = g_1 + f'\}$. Von diesen haben aber nur g_1 und g_2 einen nichttrivialen ggT mit f. Das sind jedoch genau die Elemente von $\mathcal{N} \backslash L^3$.

Die Wahrscheinlichkeit, daß der Algorithmus 2.6, angewandt auf die Niederreiter-Matrix N von f, als Ergebnis g_1 oder g_2 liefert, beträgt nach der Analyse in Abschnitt 2.2

$$\frac{1 + 2^2}{2^9 - 1} = \frac{5}{511} \sim \frac{1}{100}$$

2.4 Der Euklidische Algorithmus

Zur vollständigen Beschreibung von Wiedemanns Methode fehlt noch ein „Baustein“: die Berechnung von Minimalpolynomen linear rekurrenter Folgen über $\mathbb{F}_q$. Wie bereits in Abschnitt 2.1 angedeutet, gibt es dafür ein klassisches Verfahren nach Berlekamp und Massey ([Ber68], [Mas69]). Wir wollen hier beschreiben, wie der Euklidische Algorithmus mit ähnlicher Laufzeit zur Lösung des Problems herangezogen werden kann. Das hat den Vorteil, daß in einer Implementierung des auf Wiedemanns Ansatz beruhenden Faktorisierungsverfahrens auf Standard-Algorithmen für Polynomarithmetik zurückgegriffen werden kann. Die Idee dafür geht zurück auf Sugiyama et al. ([SKHN75]).

Zunächst noch einige theoretische Hilfsmittel. Sei $\mathbb{F}$ ein beliebiger Körper und $\mathbb{F}^\infty$ die Menge aller abzählbaren Folgen über $\mathbb{F}$ wie in Abschnitt 2.1.

Definition 2.9 (erzeugende Funktion) *Die formale Reihe*

$$a(X) := \sum_{i\geq 0} a_i X^i \in \mathbb{F}[[X]]$$

heißt erzeugende Funktion *der Folge* $a = (a_i)_{i\geq 0} \in \mathbb{F}^\infty$.

Aus den Definitionen in Abschnitt 2.1 folgt, daß die Folge a genau dann linear rekurrent ist (mit charakteristischem Polynom $f = \sum_{0\leq i\leq n} f_i X^i \in \mathbb{F}[X]$, o.E. $f_n = 1$), wenn sich ihre erzeugende Funktion $a(X)$ als *rationale Funktion* schreiben läßt:

$$a(X) = \frac{g}{f^*} \text{ für ein } g \in \mathbb{F}[X] \text{ mit } \deg g < n \tag{2.4}$$

Dabei bezeichnet $f^* := X^{\deg f} \cdot f(\frac{1}{X}) = \sum_{0\leq i\leq n} f_{n-i} X^i$ das zu f *reziproke Polynom.* Beachte: Wegen $f^* = (X^e f)^*$ für alle $e \in \mathbb{N}$ können sich f und $(f^*)^*$ um eine X-Potenz unterscheiden! Da der Leitkoeffizient bei der *-Transformation zum konstanten Koeffizienten wird, ist X kein Teiler von f^*. Ferner ist $(f^*)^* = \frac{f}{X^e}$, wobei X^e die höchste X-Potenz ist, die f teilt. Also gilt stets $(f^*)^* | f$ mit Gleichheit genau dann, wenn X kein Teiler von f, d.h. $f(0) \neq 0$ ist.

Lemma 2.10 *Sei $a \in \mathbb{F}^\infty$ linear rekurrent mit charakteristischem Polynom f wie oben. Dann ist f genau dann das Minimalpolynom von a, wenn in der Darstellung (2.4) $ggT(f^*, g) = 1$ und $\deg f = \max(\deg f^*, \deg g + 1)$ gilt.*

Beweis:

„$\Rightarrow$“ Sei zunächst f das Minimalpolynom von a. Angenommen, $d := \mathrm{ggT}(f^*, g)$ mit $\deg d \geq 1$, dann hat $a(X)$ eine „kürzere“ Darstellung als rationale Funktion $a(X) = \frac{g/d}{f^*/d}$. Das Polynom d ist als Teiler von f^* nicht durch X teilbar, also gilt $(d^*)^* = d$, und $\frac{f}{d^*}$ ist ein generierendes Polynom für a kleineren Grades als f, im Widerspruch zur Minimalität von f.

Stets gilt $\deg f \geq \deg f^*$ und nach (2.4) $\deg f > \deg g$, da f ein generierendes Polynom von a ist. Falls $\deg f = \deg f^* + e$ für ein $e \geq 1$, so ist $f = X^e (f^*)^*$. Aufgrund der Minimaleigenschaft von f ist $\overline{f} := X^{e-1}(f^*)^*$ *kein* charakteristisches Polynom für a. Wegen $a = \frac{g}{f^*} = \frac{g}{\overline{f}^*}$ muß daher nach (2.4) $\deg g \geq \deg \overline{f} = \deg f - 1$ sein. Zusammen mit $\deg g < \deg f$ folgt $\deg f = \deg g + 1$.

„$\Leftarrow$“ Sei andererseits $\mathrm{ggT}(f^*, g) = 1$, $\deg f = \max(\deg f^*, \deg g + 1)$ und f_1 das Minimalpolynom von a. Dann hat $a(X)$ zwei Darstellungen als rationale Funktion:

$$\frac{g}{f^*} = a(X) = \frac{g_1}{{f_1}^*}$$

für ein geeignetes Polynom $g_1 \in \mathbb{F}[X]$ mit $\deg f_1 = \max(\deg {f_1}^*, \deg g_1 + 1)$. Daraus folgt jedoch $g{f_1}^* = g_1 f^*$ in $\mathbb{F}[X]$, und da f^* und g relativ prim sind, muß f^* ein Teiler von ${f_1}^*$ sein. Damit gilt $(f^*)^* | ({f_1}^*)^* | f_1 | f$ (wegen der Minimalität von f_1), also $f = f_1 \cdot X^e$ für ein $e \in \mathbb{N}$ und $f^* = {f_1}^*$, $g = g_1$. Aus den Gradbedingungen ergibt sich

$$\begin{aligned} \deg f &= \max(\deg f^*, \deg g + 1) = \max(\deg {f_1}^*, \deg g_1 + 1) \\ &= \deg f_1 \end{aligned}$$

und damit $f = f_1$.

□

Wir werden sehen, daß der Euklidische Algorithmus, angewandt auf X^{2n} und $a(X) \bmod X^{2n}$, das Minimalpolynom der Folge a berechnet, falls eine obere Schranke n für die Rekursionsordnung (d.h. den Grad des Minimalpolynoms) von a bekannt ist.

Algorithmus 2.11 (Minimalpolynom mit Euklid)

{ *input: Schranke $n \in \mathbb{N}$ für die Rekursionsordnung, sowie die ersten $2n$ Folgenglieder $a_0, \ldots, a_{2n-1}$ der linear rekurrenten Folge $a \in \mathbb{F}^\infty$* }

{ *output: Das Minimalpolynom m_a von a* }

I. $r_{-1} := X^{2n}$; $r_0 := \sum_{0 \leq j < 2n} a_j X^j$;
$s_{-1} := 1$; $s_0 := 0$;
$t_{-1} := 0$; $t_0 := 1$;

II. Wiederhole Schritt III. für $i = 0, 1, \ldots$ *so lange, bis* $\deg r_i < n$:

III. $q_{i+1} := \lfloor \frac{r_{i-1}}{r_i} \rfloor$; { *Quotient bei Division mit Rest* }
$r_{i+1} := r_{i-1} - q_{i+1} r_i$;
$s_{i+1} := s_{i-1} - q_{i+1} s_i$;
$t_{i+1} := t_{i-1} - q_{i+1} t_i$;

IV. $e := \max(\deg r_i + 1 - \deg t_i^*, 0)$;

V. output $X^e t_i^*$.

Bemerkung

1. Wie die Abbruchbedingung II. zeigt, wird der Euklidische Algorithmus nicht bis zum Ende ($r_i = 0$), sondern nur bis zur „Hälfte“ iteriert.
2. Die s_i sind eigentlich für diese Berechnung überflüssig, werden aber für den Korrektheitsnachweis benötigt.

Korrektheitsbeweis: Die r_i bilden wegen $\deg r_0 < 2n$ und der Divisionseigenschaft ($\deg r_{i+1} < \deg r_i$) eine bezüglich des Grades echt absteigende Folge, d.h. die Terminierungsbedingung $\deg r_i < n$ wird nach höchstens n Iterationen erreicht. Außerdem ist $\deg q_i \geq 1$ für alle $i \geq 1$ (bis zum Abbruch der Schleife II.).

Bezeichne $b := r_0 = \sum_{0 \leq j < 2n} a_i X^i = a(X) \bmod X^{2n}$ die $\bmod X^{2n}$-Approximation von $a(X)$. Wir weisen nach, daß für alle $i = 0, 1, \ldots$ die Schleifeninvarianten

i. $s_i X^{2n} + t_i b = r_i$ (auch für $i = -1$)

ii. $s_{i-1} t_i - s_i t_{i-1} = (-1)^i$

iii. $\deg t_i = \sum_{1 \leq k \leq i} \deg q_k$

iv. $\deg r_i = 2n - \deg t_{i+1}$ (auch für $i = -1$)

erfüllt sind. Der Induktionsanfang $i = -1, 0$ folgt direkt aus den Zuweisungen in Schritt I. Für den Induktionsschritt nehmen wir die Aussagen für alle $k \leq i$ bereits als bewiesen an und zeigen, daß sie dann auch für $i + 1$ gelten:

$$\begin{aligned} s_{i+1}X^{2n} + t_{i+1}b &= (s_{i-1} - q_{i+1}s_i)X^{2n} + (t_{i-1} - q_{i+1}t_i)b \\ &= (s_{i-1}X^{2n} + t_{i-1}b) - q_{i+1}(s_iX^{2n} + t_ib) \\ &= r_{i-1} - q_{i+1}r_i = r_{i+1} \end{aligned}$$

$$\begin{aligned} s_it_{i+1} - s_{i+1}t_i &= s_i(t_{i-1} - q_{i+1}t_i) - (s_{i-1} - q_{i+1}s_i)t_i \\ &= -(s_{i-1}t_i - s_it_{i-1}) = -(-1)^i = (-1)^{i+1} \end{aligned}$$

$$\begin{aligned} \deg t_{i+1} &= \deg(t_{i-1} - q_{i+1}t_i) \\ &= \deg q_{i+1} + \deg t_i, \text{ da per Induktion } \deg t_{i-1} < \deg t_i \\ &= \deg q_{i+1} + \sum_{1\le k\le i} \deg q_k = \sum_{1\le k\le i+1} \deg q_k \end{aligned}$$

$$\begin{aligned} \deg r_{i+1} &= \deg r_i - \deg q_{i+2} = 2n - (\deg t_{i+1} + \deg q_{i+2}) \\ &= 2n - \deg t_{i+2} \text{ nach iii.} \end{aligned}$$

Sei $l \geq 0$ der kleinste Index, für den die Terminierungsbedingung erfüllt ist. Dann gilt:

1. $2n - \deg t_l = \deg r_{l-1} \geq n \Rightarrow \deg t_l \leq n$
2. $\deg r_l < n$
3. $s_lX^{2n} + t_lb = r_l$, d.h. $t_lb \equiv r_l \bmod X^{2n}$
4. $s_{l-1}t_l - s_lt_{l-1} = (-1)^l$

Sei e wie in Schritt IV. des Algorithmus, $f := X^e {t_l}^*$ und $g := \frac{r_l}{\mathrm{ggT}(r_l,t_l)}$. Um zu zeigen, daß f das Minimalpolynom von a ist, genügt es, die Bedingungen von Lemma 2.10 nachzuweisen.

a. $\mathrm{ggT}(f^*, g) = 1$: Nach 3. gilt $\mathrm{ggT}(t_l, r_l) | (s_lX^{2n})$, wegen 4. sind s_l und t_l teilerfremd, d.h. $\mathrm{ggT}(t_l, r_l) = X^d$ für ein $d \in \mathbb{N}$, $d < n$. Ebenfalls nach 3. ist X^{d+1} kein Teiler von t_l, d.h. $f^* = ({t_l}^*)^* = \frac{t_l}{X^d} = \frac{t_l}{\mathrm{ggT}(r_l,t_l)}$ und $\mathrm{ggT}(f^*, g) = \mathrm{ggT}(\frac{t_l}{X^d}, \frac{r_l}{X^d}) = \frac{\mathrm{ggT}(t_l,r_l)}{X^d} = 1$.

b. $a(X) = \frac{g}{f^*}$: Da a linear rekurrent mit Rekursionsordnung $\leq n$ ist, gibt es $f_1, g_1 \in \mathbb{F}[X]$ mit $\deg g_1 < \deg f_1 \leq n$ und $a(X) = \frac{g_1}{f_1{}^*}$. Diese Gleichung gilt auch mod X^{2n}, d.h. $f_1{}^* b \equiv f_1{}^* a(X) \equiv g_1 \pmod{X^{2n}}$. Multiplizieren wir diese Kongruenz mit t_l und die Kongruenz 3. mit $f_1{}^*$, so erhalten wir

$$t_l g_1 \equiv t_l b f_1{}^* \equiv r_l f_1{}^* (\mathrm{mod}\ X^{2n})$$

Da die beiden Polynome $t_l g_1$ und $r_l f_1{}^*$ nach 1. und 2. echt kleineren Grad als $2n$ haben, sind sie gleich, d.h. $a(X) = \frac{g_1}{f_1{}^*} = \frac{r_l}{t_l} = \frac{g}{f^*}$ (siehe a.).

c. $\deg f = \max(\deg f^*, \deg g + 1)$: folgt sofort aus den Definitionen.

$\square$

Die Kosten für Algorithmus 2.11 sind $\mathcal{O}(n^2)$ Körperoperationen in $\mathbb{F}$, denn: Die Grade aller vorkommenden Polynome r_i, s_i, t_i sind höchstens $2n$. Die Berechnungen in Schritt III. erfordern $\mathcal{O}(n \cdot \deg q_{i+1})$ Operationen in $\mathbb{F}$ (mit klassischer Arithmetik). Die Gesamtkosten ergeben sich damit als $\sum_{0 \leq i \leq l} \mathcal{O}(n \cdot \deg q_{i+1}) = \mathcal{O}(n \cdot \deg t_{l+1}) = \mathcal{O}(n^2)$. Die asymptotische Laufzeit des Verfahrens nach Berlekamp / Massey liegt in der gleichen Größenordnung.

Ein „divide and conquer“-Ansatz des Euklidischen Algorithmus führt zu einem $\mathcal{O}(M(n) \log n)$-Algorithmus für die ggT-Berechnung (vgl. [Moe73], [AHU75] Kap. 8). Derselbe Ansatz führt auch für Algorithmus 2.11 zum Erfolg und liefert ein Verfahren zur Minimalpolynom-Berechnung mit Laufzeit $\mathcal{O}(M(n) \log n)$ (vgl. [BGY80]). Dies bringt allerdings asymptotisch keinen Gewinn für die Laufzeit von Wiedemanns Methode, da deren „bottleneck“ nicht die Berechnung von Minimalpolynomen über $\mathbb{F}_q$ ist, sondern die „black box“-Anwendung der Matrix.

Zum Abschluß sei für einen Vergleich der Algorithmen von Euklid und Berlekamp / Massey auf die Literatur verwiesen ([WS79], [BGY80], [Bla85], [Dor87], [Bla92]). Die Beschreibung einer „divide and conquer“-Version des letzteren mit einer Laufzeit von ebenfalls $\mathcal{O}(M(n) \log n)$ ist in [BGY80] und [Bla85] zu finden.

Kapitel 3

Implementation und Programmdokumentation

Im Rahmen dieser Diplomarbeit wurden die Algorithmen von Berlekamp (Algorithmus 1.4) in der *absoluten* Version (vgl. Bem. 1.7), Niederreiter (Algorithmus 1.11) und von Kaltofen & Lobo (Algorithmus 2.7) für den (absoluten) Berlekamp- und den Niederreiter-Fall (vgl. Abschnitt 2.3) in Charakteristik 2 implementiert. Die verwendete Programmiersprache ist C++ (vgl. [ES90]).

Der Grund dafür, daß die absolute statt der relativen Version des Berlekamp-Verfahrens gewählt wurde, ist die Analogie zu Niederreiters Verfahren. Viele Programmteile konnten bis auf den Austausch der Matrizen $\overline{B}$ und N bzw. der zugehörigen „black box"-Operatoren identisch gehalten werden. Für die Version von Berlekamps Algorithmus, in welcher die „absolute" Matrix $\overline{B}$ Gauß-eliminiert wird, bedeutet das eine – zumindest asymptotisch – längere Laufzeit und erhöhten Speicherbedarf gegenüber der relativen Version (vgl. Bem. 1.7). Diese Differenzen bestehen beim Algorithmus von Kaltofen & Lobo nicht, wie bereits in Abschnitt 2.3 angedeutet.

Das Programm führt Buch über die verbrauchten Ressourcen (Speicherplatz und CPU-Zeit) und mißt die Anzahl Körperoperationen (allerdings nicht in $\mathbb{F}_2$, weil das die Laufzeit nicht unerheblich beeinträchtigen würde, sondern in $\mathbb{F}_q$) für einen Durchlauf. Es ist möglich, den Programmablauf protokollieren zu lassen. Die Daten einiger Testläufe sind in Abschnitt 3.4 zu finden. Sie machen leider deutlich, daß das Programm in „vernünftiger" Zeit (1 Tag CPU-Zeit) nur Input-Größenordnungen (Körpergrad über $\mathbb{F}_2$ mal Polynomgrad) bis zu 2000 verarbeiten kann. Das ist allerdings kein prinzipielles Problem der verwendeten Algorithmen, sondern darauf zurückzuführen, daß die zugrundeliegende Körperarithmetik in $\mathbb{F}_q$ nicht auf Geschwindigkeit hin optimiert wurde. So verbraucht z.B. ein Element von $\mathbb{F}_2$ ein Byte Speicherplatz; würden 8 Elemente aus $\mathbb{F}_2$ in ein Byte gepackt, dann würde das nicht nur den Speicherbedarf verringern, sondern auch die Laufzeit.

Als schneller Multiplikationsalgorithmus, sowohl für Elemente von $\mathbb{F}_q$ als auch für Polynome über $\mathbb{F}_q$, ist die Karatsuba-Multiplikation mit einer Laufzeit von $M(n) = \mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.59})$ für die Multiplikation zweier Polynome vom Grad n implementiert. Die Testdaten zeigen allerdings, daß in dieser Implementation die Faktorisierungsverfahren, die auf Wiedemanns Methoden unter Verwendung der Karatsuba-Multiplikation basieren, für die „machbaren" Größenordnungen noch nicht in der Lage sind, die klassischen Algorithmen bezüglich der tatsächlichen Laufzeit „auszustechen". Asymptotisch – möglicherweise auch praktisch – ist hier noch eine Verbesserung möglich, wenn die FFT-Multiplikation als schnelle Multiplikation implementiert wird.

3.1 Datentypen

C++ bietet zwei Konzepte, die beim Entwurf der Datentypen für die Faktorisierungsalgorithmen gute Verwendung fanden: Vererbung und Templates. In unserer Situation treten verschiedene Arten von Daten auf: Körperelemente von $\mathbb{F}_2$ und $\mathbb{F}_q$ und Polynome mit Koeffizienten aus diesen beiden Körpern.

Polynome haben gewisse Eigenschaften, die unabhängig vom jeweiligen Koeffizientenbereich sind: etwa der Grad oder die Vorschrift, wie zwei Polynome über demselben Koeffizientenbereich zu addieren (koordinatenweise) oder zu multiplizieren (klassisch Faltung, oder „divide and conquer" (Karatsuba)) sind. Ein Unterschied besteht lediglich auf der Ebene der Koeffizienten bzgl. deren Speicherbedarf und Arithmetik. Dieser strukturellen Ähnlichkeit wird das Konzept der *Templates* gerecht. Ein Template ist ein Datentyp (eine Klasse) oder eine Funktion mit einem abstrakten Datentyp T als Parameter, über den beim Entwurf des Templates nichts bekannt zu sein braucht. In unserer Situation bedeutet das, daß ein Datentyp „Polynom mit abstrakten Koeffizienten aus T" und arithmetischen Operationen auf diesen Polynomen definiert werden kann, von dem später *Instanzen* für konkrete Koeffizienten-Typen (z.B. für $T = \mathbb{F}_2$ oder $T = \mathbb{F}_q$) automatisch abgeleitet werden können. Das heißt, daß z.B. für die Addition von Polynomen nur eine Prozedur definiert werden muß, die sowohl für $\mathbb{F}_2$-Polynome als auch für $\mathbb{F}_q$-Polynome verwendet werden kann.

Weiterhin bestehen Gemeinsamkeiten zwischen $\mathbb{F}_2$-Polynomen und Körperelementen aus $\mathbb{F}_q$ bezüglich der Darstellung im Rechner und der Arithmetik. Elemente von $\mathbb{F}_q = \mathbb{F}_{2^r}$ werden natürlicherweise repräsentiert als Polynome $\in \mathbb{F}_2[X]$ vom Grad kleiner als r mit arithmetischen Operationen modulo einem irreduziblen Polynom $F \in \mathbb{F}_2[X]$ vom Grad r. Dabei besteht z.B. kein Unterschied zwischen der Addition von derart repräsentierten Körperelementen und der Addition von Polynomen $\in \mathbb{F}_2[X]$ (wohl aber natürlich bei der Multiplikation). Hier kommt die *Vererbung* zum Einsatz: Elemente von $\mathbb{F}_q$ werden als *spezielle* Polynome über $\mathbb{F}_2$ aufgefaßt, die einen Teil der Datenstruktur (Größe, Koeffizientenvektor, „Grad") und der Operationen des Datentyps „$\mathbb{F}_2$-Polynom" (Addition, Subtraktion) erben, einen weiteren Teil umdefinieren (Multiplikation) und zusätzliche Operationen haben (Inversion).

Der einfachste vorkommende Datentyp ist `GF2` für die Elemente von $\mathbb{F}_2$. Der Speicherplatz für ein `GF2`-Objekt beträgt 1 Byte, die wichtigsten angebotenen Operationen sind `+`, `-`, `*` und `/`.

Der Datentyp `Poly`

Die wesentliche Datenstruktur definiert der Datentyp `RawPoly`:

```
/* Klasse RawPoly: Polynome mit Koeffizienten aus T,
   aber noch ohne arithmetische Operationen */
template<class T> class RawPoly: public Algebraic {
  /* Daten */
 protected:
  index_t len;
  index_t deg;
  T*      coeff;
  /* Methoden */
  ...
 public:
  index_t degree(void) const;
  index_t ldegree(void) const;
  T&      operator [](index_t k);
  T&      lcoeff(void);
  boolean iszero(void) const;
  boolean isconst(void) const;
  ...
};
boolean operator ==(const RawPoly<T>& z1, const RawPoly<T>& z2);
boolean operator !=(const RawPoly<T>& z1, const RawPoly<T>& z2);
```

Die Klasse `Algebraic`, von der `RawPoly` abgeleitet ist, enthält lediglich Verwaltungsinformation. `index_t` ist ein Integer-Typ für die Indizierung von Koeffizienten; in der Implementation beträgt sein Wertebereich $-2^{31}..(2^{31}-1)$. `boolean` ist ein Datentyp für Wahrheitswerte.

Die Variable `len` gibt die Länge des Polynoms an, d.h. die Anzahl gespeicherter Koeffizienten, während `deg` den tatsächlichen Grad des Polynoms beinhaltet (mit der Vereinbarung, daß das Nullpolynom Grad -1 hat). `coeff` schließlich ist ein Zeiger auf das Koeffizienten-Array, d.h. `coeff[0]` bis `coeff[deg]` enthalten die Koeffizienten des Polynoms.

Die Methoden `degree` und `ldegree` liefern den Grad des Polynoms bzw. den Index des frühesten, nicht verschwindenden Koeffizienten („low degree") zurück. Der Operator `[]` gestattet den Zugriff auf die einzelnen Koeffizienten des Polynoms*, er überprüft auch, ob der Index `k` im erlaubten Bereich 0..`deg` liegt. `lcoeff` liefert den Leitkoeffizienten zurück. Die beiden Methoden `iszero` und `isconst` ergeben den Wahrheitswert `TRUE`, falls das Polynom das Nullpolynom bzw. ein konstantes Polynom ist.

Die beiden Nicht-Methoden† `==` und `!=` überprüfen zwei Polynome auf Gleichheit bzw. Ungleichheit. `RawPoly` enthält noch weitere Methoden für Verwaltungszwecke, auch Funktionen für die Ein- und Ausgabe von Objekten stehen zur Verfügung.

Von `RawPoly` abgeleitet ist die Klasse `Poly`, welche zusätzlich die arithmetischen Operationen für Polynome implementiert.

```
/* Klasse Poly: Polynome mit Koeffizienten aus T */
template<class T> class Poly: public RawPoly<T> {
  /* Daten */
  ...
  /* Methoden */
 protected:
  /* Hilfsfunktionen fuer Verwaltung und Arithmetik */
  ...
 public:
  /* Konstruktoren und Destruktor */
  Poly(void);
  Poly(const T& z);
  Poly(index_t k);
  ~Poly(void);
  ...
  /* Zuweisungsoperatoren */
  Poly<T>& operator =(const RawPoly<T>& z);
  Poly<T>& operator =(const T& z);
  Poly<T>& operator =(index_t k);
  ...
};
```

*Nach der Philosophie des „information hiding" darf auf die Daten einer Klasse nur über ihre Methoden zugegriffen werden, welche einen konsistenten Zustand der Daten garantieren.

†Das für die Deklaration von Template-Funktionen nötige `template<class T>` wird der Übersichtlichkeit halber weggelassen.

```
/* Arithmetische Operationen */
Poly<T>& random(Poly<T>& ret, index_t size);
Poly<T>& null(Poly<T>& ret, index_t size);
Poly<T>& derive(Poly<T>& ret);
Poly<T>& operator +=(Poly<T>& ret, const RawPoly<T>& z);
Poly<T>  operator +(const RawPoly<T>& z1, const RawPoly<T>& z2);
Poly<T>& operator -=(Poly<T>& ret, const RawPoly<T>& z);
Poly<T>  operator -(const RawPoly<T>& z1, const RawPoly<T>& z2);
Poly<T>& operator *=(Poly<T>& ret, const RawPoly<T>& z);
Poly<T>  operator *(const RawPoly<T>& z1, const RawPoly<T>& z2);
Poly<T>  square(const RawPoly<T>& z);
int      div(const RawPoly<T>& a, const RawPoly<T>& b,
             Poly<T>& q, Poly<T>& r);
void     ggt(const RawPoly<T>& a, const RawPoly<T>& b,
             Poly<T>& g, Poly<T>* s, Poly<T>* t);
Poly<T>& operator *=(Poly<T>& ret, const T& z);
Poly<T>& operator *=(Poly<T>& ret, index_t k);
Poly<T>  operator >>(const RawPoly<T>& z, index_t k);
Poly<T>& operator >>=(Poly<T>& ret, index_t k);
Poly<T>  reverse(const RawPoly<T>& z, index_t formal_degree);
void     split01(const RawPoly<T>& z, Poly<T>* z0, Poly<T>* z1);
...
```

Die Klasse `Poly` erbt sämtliche Daten und Methoden von `RawPoly`. Sie enthält weitere Daten und Methoden für Verwaltung und Buchhaltung (Operationen zählen), die der Übersichtlichkeit halber nicht aufgeführt sind. Die drei Konstruktoren erzeugen das Nullpolynom, ein konstantes Polynom mit konstantem Term `z` bzw. ein konstantes Polynom mit konstantem Term `k` mod 2, der Charakteristik. Weiterhin gibt es einen Operator für die Zuweisung eines Polynoms an ein anderes („copy operator") und zwei Operatoren für die Zuweisung einer Konstante an ein Polynom[‡].

Die arithmetischen Operationen sind als normale Funktionen implementiert, nicht als Methoden. Sie greifen auf Hilfs-Methoden zurück, die nur für diese Funktionen zugänglich sind. Ein Grund dafür, daß die arithmetischen Operationen nicht als Methoden konzipiert sind, ist „kosmetischer" Natur. Es erscheint wenig sinnvoll, eine Methode `Poly<T> operator +(const Poly<T>& z)` zu definieren, deren Semantik darin besteht, daß ein Objekt (Polynom) sich selbst zu einem anderen Objekt (Polynom) addiert und das Ergebnis in ein drittes Objekt kopiert. Vielmehr ist hier die „symmetrische" und nicht objektbezogene Semantik erwünscht, daß zwei gleichberechtigte Objekte (Polynome) addiert werden und das Ergebnis in einem dritten Objekt zurückgeliefert wird. Deshalb bietet die Klasse `Poly` Hilfs-Methoden für die Arithmetik an, die die eigentlichen Algorithmen implementieren, aber nicht nach außen sichtbar sind. Darauf aufgesetzt sind als Export-Schnittstelle die aufgeführten Nicht-Methoden.

[‡] Der Unterschied zwischen Konstruktion und Zuweisung liegt darin, daß bei der Zuweisung der vorherige Inhalt zerstört werden muß, bei der Konstruktion gibt es per Definition keinen vorherigen Inhalt.

Die meisten arithmetischen Operationen sind selbsterklärend. Ein Operator gefolgt von einem Zuweisungszeichen `=` verknüpft, wie bei den C-Grundoperationen üblich, das erste Argument des Operators mit dem zweiten und schreibt das Ergebnis in das erste Argument. Der Befehl `a += b` etwa ist äquivalent zu `c = a + b; a = c`. Alle Funktionen mit dem Rückgabetyp `Poly<T>&` liefern ihr erstes Argument (genauer gesagt eine Referenz darauf) zurück, diejenigen mit dem Rückgabetyp `Poly<T>` legen ihr Ergebnis in einem neuen Objekt ab.

`random` schreibt ein Polynom mit `size` zufällig und unabhängig gewählten Koeffizienten in sein erstes Argument. `null` macht aus seinem ersten Argument ein Nullpolynom mit `size` Koeffizienten (die natürlich alle gleich 0 sind). `derive` berechnet die formale Ableitung seines Arguments. Die Subtraktions-Operatoren `-` und `-=` sind der Vollständigkeit halber aufgeführt, sie haben in Charakteristik 2 natürlich dieselbe Semantik wie `+` bzw. `+=`. `div` führt eine Division von `a` durch `b` mit Rest durch. Der Quotient wird in `q` abgelegt und der Rest in `r`. Die Methode `ggt` berechnet den normierten größten gemeinsamen Teiler `g` von `a` und `b` einschließlich der Koeffizienten `*s` und `*t` für seine Darstellung als Linearkombination von `a` und `b`. Die beiden unteren `*=`-Operatoren multiplizieren ihr erstes Argument mit einer Konstanten (Skalarmultiplikation). Die Operatoren `>>` und `>>=` sind „shift"-Operatoren, sie verschieben die Koeffizientenfolge ihres ersten Arguments um `k` (das entspricht einer Multiplikation mit $X^{\mathtt{k}}$). `reverse` kehrt die Koeffizientenfolge eines Polynoms um: $\mathtt{reverse(z)} = X^{\mathtt{formal\_degree}} \cdot \mathtt{z}(\frac{1}{X})$, berechnet also das reziproke Polynom von `z` (vgl. Abschnitt 2.4), falls in `formal_degree` der Grad von `z` übergeben wird. `split01` schließlich zerlegt `z` in zwei Polynome `*z0` und `*z1`, deren Koeffizientenfolgen die Koeffizienten von `z` mit geraden bzw. ungeraden Indizes sind.

Die verwendeten Algorithmen sind Standard und von der Laufzeit her linear, bis auf `mult`, `div` und `ggt`. Hierfür sind sowohl die klassischen quadratischen Verfahren (Faltung, „long division" bzw. Euklidischer Algorithmus) implementiert als auch asymptotisch schnellere Algorithmen, die auf der Karatsuba-Multiplikation aufbauen. Die Laufzeit für die schnelle Polynomdivision eines Polynoms vom Grad $2n$ durch ein Polynom vom Grad n mittels Newton-Inversion (vgl. [Kun74]) beträgt $\mathcal{O}(M(n)) = \mathcal{O}(n^{1.59})$. Die Zeitkomplexität für eine „divide and conquer"-Version von Euklids Algorithmus (vgl. [Moe73], [AHU75] Kap. 8 und [BGY80]) ist $\mathcal{O}(M(n)\log n) = \mathcal{O}(n^{1.59}\log n)$ für Polynome vom Grad n.

Aus dem Klassen-Template `Poly` werden die beiden Template-Instanzen `Poly2` für Polynome über $\mathbb{F}_2$ und `Poly2r` für Polynome über $\mathbb{F}_q = \mathbb{F}_{2^r}$ gebildet.

Der Datentyp `Extension`

Die Klasse `Extension` modelliert die Elemente einer Körpererweiterung, dargestellt als Polynome mit beschränktem Grad über dem Grundkörper. In der Implementation wird sie für die Elemente von $\mathbb{F}_q = \mathbb{F}_{2^r}$ verwendet.

```
/* Klasse Extension: Algebraische Erweiterung des Koerpers T */
template<class T> class Extension: public Poly<T> {
  /* Daten */
 public:
  static Extension_data<T> field;
  ...
  /* Methoden */
 protected:
  /* Hilfsfunktionen fuer Verwaltung und Arithmetik */
  ...
 public:
  /* Konstruktoren */
  Extension(void);
  Extension(Poly<T>& z);
  Extension(const T& z);
  Extension(index_t k);
  /* Zuweisungsoperatoren */
  Extension<T>& operator =(const Extension<T>& z);
  Extension<T>& operator =(const T& z);
  Extension<T>& operator =(index_t k);
};
/* Arithmetische Operationen */
Extension<T>& random(Extension<T>& ret, index_t size);
Extension<T>& null(Extension<T>& ret, index_t size);
Extension<T>& operator +=(Extension<T>& ret, const Extension<T>& z);
Extension<T>  operator +(const Extension<T>& z1,
                         const Extension<T>& z2);
Extension<T>& operator -=(Extension<T>& ret, const Extension<T>& z);
Extension<T>  operator -(const Extension<T>& z1,
                         const Extension<T>& z2);
Extension<T>& operator *=(Extension<T>& ret, const Extension<T>& z);
Extension<T>  operator *(const Extension<T>& z1,
                         const Extension<T>& z2);
Extension<T>  square(const Extension<T>& z);
Extension<T>& operator *=(Extension<T>& ret, const T& z);
Extension<T>& operator *=(Extension<T>& ret, index_t k);
Extension<T>  operator /(const Extension<T>& z1,
                         const Extension<T>& z2);
Extension<T>  operator ^(const Extension<T>& z1, index_t k);
...
```

`Extension` erbt alle Daten und (Hilfs-)Methoden von `Poly`, die nicht neu definiert werden. An Daten kommt im wesentlichen der Datensatz `field` hinzu, welcher die für alle Objekte dieser Klasse identischen Informationen (erzeugendes Polynom der Körpererweiterung, Erweiterungsgrad etc.) enthält und einmal für die gesamte Klasse global gespeichert wird. Die Konstruktoren und Zuweisungsoperatoren haben eine analoge Semantik wie bei `Poly`. Neu hinzugekommen ist ein *Konversionskonstruktor*, welcher ein Element der Körpererweiterung aus einem Polynom über dem Grundkörper konstruiert (ggf. durch Reduktion modulo dem erzeugenden Polynom).

Die Semantik einiger arithmetischer Operationen (`+=`, `+`, `-=`, `-` sowie der beiden Skalarmultiplikations-Operatoren `*=`) entspricht der Semantik der gleichnamigen `Poly`-Operationen bzw. ist nur unwesentlich abgeändert (`random` und `null` etwa lassen für `size` nur Werte kleiner als der Erweiterungsgrad zu). Die Vorteile der Vererbung sind dabei nicht direkt ersichtlich, aber der Mechanismus setzt bei den nicht näher erläuterten Hilfs-Methoden für die Arithmetik ein, die `Extension` von `Poly` erbt und auf die die arithmetischen Operatoren für die Klasse `Extension` zugreifen. Der eigentliche Algorithmus existiert dabei nur einmal (nämlich als Methode der Klasse `Poly`), wird aber in zwei Formen nach außen angeboten (nämlich für die Klassen `Poly` und `Extension`).

Die multiplikativen Operatoren `*=`, `*` und `square` multiplizieren wie bei `Poly` und reduzieren (per Division mit Rest) das Ergebnis anschließend modulo dem erzeugenden Polynom der Körpererweiterung. Neue Operationen sind der Divisionsoperator `/` und der Exponentiationsoperator `^`, welcher mittels schneller Exponentiation (sukzessives Quadrieren und Multiplizieren) sein Argument in die `k`-te Potenz erhebt.

Die aus der Template-Klasse `Extension` erzeugte Template-Instanz `GF2r` repräsentiert die Elemente von $\mathbb{F}_q = \mathrm{GF}(2^r)$.

Die Datentypen `Vector` und `Matrix`

Weitere wesentliche Datentypen sind die Template-Klassen `Vector` und `Matrix` für Vektoren bzw. Matrizen mit Einträgen aus einem Körper. Die daraus erzeugten Template-Instanzen sind `Vector2` und `Matrix2` für Vektoren bzw. Matrizen über $\mathbb{F}_2$.

```
template<class T> class Vector {
  /* Daten */
 protected:
  index_t len;
  T*      coeff;
```

```
  /* Methoden */
 public:
  /* Konstruktor und Destruktor */
  Vector(index_t n);
  ~Vector(void);
  /* Zuweisungsoperator */
  Vector<T>& operator =(const Vector<T>& v);
  index_t    dim(void) const;
  T&         operator [](index_t i);
};
/* Klasse Matrix: realisiert als Array von Zeilen-Vektoren */
template<class T> class Matrix {
  /* Daten */
 protected:
  index_t     rows;
  index_t     cols;
  Vector<T>** coeff;
  /* Methoden */
  ...
 public:
  /* Konstruktoren und Destruktor */
  Matrix(index_t r, index_t c);
  Matrix(index_t n);
  ~Matrix(void);
  /* Zuweisungsoperator */
  Matrix<T>& operator =(const Matrix<T>& A);
  index_t    rowdim(void) const;
  index_t    coldim(void) const;
  Vector<T>& operator [](index_t i);
};
/* Arithmetische Operationen */
Matrix<T>* nullspace(const Matrix<T>& A);
...
```

Zunächst zum Datentyp `Vector`: Die Variable `len` enthält die Länge des Vektors, und `coeff` zeigt auf das Koeffizienten-Array. Der Konstruktor erzeugt einen Nullvektor der Dimension `n`. Der Zuweisungsoperator übernimmt den Inhalt des Vektors `v`. Die Methode `dim` liefert die Dimension zurück, und der Operator `[]` gestattet den (kontrollierten) Zugriff auf die Koeffizienten.

Die Klasse `Matrix` enthält als Daten die Variablen `rows` und `cols` für die Zeilen- bzw. Spaltendimension, sowie einen Zeiger `coeff` auf das Array der Zeilenvektoren (die doppelte Verzeigerung hat technische Gründe). Der zweistellige Konstruktor erzeugt eine mit Nullen gefüllte Matrix mit `r` Zeilen und `c` Spalten, der einstellige eine `n`-dimensionale quadratische Nullmatrix. Der Zuweisungsoperator verhält sich wie bei `Vector`. Die Methoden `rowdim` und `coldim` liefern die Zeilen- bzw. Spaltendimension der Matrix zurück. Der Zugriffsoperator `[]` liefert eine

Referenz auf den `i`-ten Zeilenvektor zurück, d.h. auf das `j`-te Element in Zeile `i` einer Matrix `A` kann mit `A[i][j]` zugegriffen werden.

Die Template-Funktion `nullspace` berechnet durch Gauß-Elimination eine Basis des (rechten) Nullraums der Matrix und liefert diese in Form einer neuen Matrix von Zeilenvektoren zurück, d.h. deren Zeilendimension ist die Dimension des Nullraums. Für die Elimination einer $r \times c$-Matrix A wird diese um eine c-dimensionale Einheitsmatrix I nach „unten" erweitert (d.h. die erweiterte Matrix hat die Form $\left[\begin{smallmatrix} A \\ I \end{smallmatrix}\right]$). Dann wird spaltenweise eliminiert, wobei allerdings nur im oberen Teil der Matrix pivotiert werden darf. Am Schluß der Elimination bilden diejenigen Spalten im unteren Teil, bei denen im oberen Teil eine Nullspalte steht, eine Basis des Nullraums von A (vgl. [MV90] p. 294).

Der Datentyp `BlackBox`

Für die „black box"-Repräsentation einer quadratischen Matrix steht der Datentyp `BlackBox` zur Verfügung. Es wird davon ausgegangen, daß der zugrundeliegende Vektorraum als Menge von Polynomen beschränkten Grades über dem Grundkörper repräsentiert und eine „black box"-Anwendung der Matrix durch Polynomarithmetik berechnet werden kann. Das ist sowohl bei Berlekamp als auch bei Niederreiter der Fall.

```
/* Abstrakte Klasse BlackBox: Black Box-Repraesentation eines
   linearen Operators, bietet als wesentliche Operation die
   Anwendung des Operators auf einen Vektor an, sowie einen Puffer,
   in dem das jeweils zuletzt berechnete Ergebnis steht.
*/
template<class T> class BlackBox {
  /* Daten */
 protected:
  index_t dim;
  Poly<T> buf[2];
  int     bufptr;
  ...
  /* Methoden */
 public:
  /* Konstruktor */
  BlackBox(index_t d);
  index_t      dimension(void);
  Poly<T>&     buffer(void);
  Poly<T>&     oldbuffer(void);
  virtual void applyto(const Poly<T>& h, Poly<T>& ret) = 0;
  void         applyto_buffer(void);
};
```

Die Variable `dim` enthält die Dimension der Matrix. `buf` ist ein Puffer für zwei Vektoren (Polynome) mit Koeffizienten aus `T`. Er erweist sich als zweckmäßig, wenn der „black box"-Operator *iteriert* auf einen Vektor angewandt werden soll, wie das bei der Berechnung von Minimalpolynomen mit Wiedemanns Methode der Fall ist. Der „ältere" Teil des Puffers (`buf[1-bufptr]`) enthält dabei den Vektor, auf den der Operator angewandt wird, und der „neuere" Teil (`buf[bufptr]`) das Ergebnis der Anwendung.

Der Konstruktor erzeugt einen „black box"-Operator der Dimension `d`. Mit `dimension` kann die Dimension abgefragt werden, `buffer` und `oldbuffer` liefern eine Referenz auf die jeweiligen Teile des Puffers zurück.

Der Kern von `BlackBox` ist die Methode `applyto`. Sie ist als sogenannte *„pure virtual function"* implementiert (vgl. [ES90]), d.h. sie muß für die Klasse `BlackBox` noch nicht und kann in jeder von `BlackBox` abgeleiteten Klasse individuell definiert werden. Sie berechnet eine Anwendung des „black box"-Operators auf `h` und legt das Ergebnis in `ret` ab. `applyto_buffer` wendet den Operator auf den im Puffer befindlichen Vektor an und schreibt das Ergebnis in den anderen Teil des Puffers.

Von `BlackBox` sind zwei Klassen abgeleitet, `Berlekamp` und `Niederreiter`.

```
/* Klasse Berlekamp: Black Box-Version des absoluten
   Berlekamp-Operators */
class Berlekamp : public BlackBox<GF2r> {
  /* Daten */
 protected:
  const Poly2r& f;
  /* Methoden */
 public:
  /* Konstruktor */
  Berlekamp(const Poly2r& z);
  virtual void applyto(const Poly2r& h, Poly2r& ret);
};
/* Klasse Niederreiter: Black Box-Version des Niederreiter-Operators */
class Niederreiter : public BlackBox<GF2r> {
  /* Daten */
 protected:
  const Poly2r& f;
  /* Methoden */
 public:
  /* Konstruktor */
  Niederreiter(const Poly2r& z);
  virtual void applyto(const Poly2r& h, Poly2r& ret);
};
```

Beide Klassen enthalten gegenüber `BlackBox` eine neue Variable `f`, die das zu faktorisierende Polynom aus $\mathbb{F}_q[X]$ enthält. Sie wird von den Konstruktoren initialisiert. Die Methode `applyto` ist gemäß den Abbildungsvorschriften für die Operatoren $\overline{\beta}$ bzw. ν in Abschnitt 2.3 implementiert. Bei der Implementation von ν ist noch eine Beschleunigung möglich (vgl. [GvzG94]): Für ein Polynom $h \in \mathbb{F}_q[X]$ bezeichne h_0 bzw. h_1 seinen geraden bzw. ungeraden Anteil, d.h. $h = h_0(X^2) + Xh_1(X^2)$. Dann ist

$$\begin{aligned} \nu h &= ((hf)' + h^2)(\sqrt{X}) = ((h_0 f_1 + h_1 f_0)(X^2) + h^2)(\sqrt{X}) \\ &= h_0 f_1 + h_1 f_0 + h^2(\sqrt{X}) \end{aligned}$$

d.h. statt einer Multiplikation zweier Polynome vom Grad $\leq n$ sind für eine „black box"-Anwendung von ν im wesentlichen zwei Multiplikationen von Polynomen vom Grad $\leq \frac{n}{2}$ erforderlich. Bei Verwendung der klassischen Multiplikation verringert das die Rechenzeit für eine Anwendung von ν ungefähr um den Faktor $\frac{1}{2}$.

Bemerkung Die Klassen `Berlekamp` und `Niederreiter` implementieren zwar „black box"-Operatoren über $\mathbb{F}_q$, stellen aber tatsächlich keine $\mathbb{F}_q$-linearen, sondern nur $\mathbb{F}_2$-lineare Abbildungen dar. Insbesondere ist die Dimension des Problems gleich dem Produkt aus dem Grad des Polynoms `f` (über $\mathbb{F}_q$) und dem Erweiterungsgrad r von $\mathbb{F}_q$ über $\mathbb{F}_2$. Der Grund für diese etwas inkonsequente Darstellung ist, daß dadurch lästige Typkonversionen von $\mathbb{F}_q$-Polynomen in $\mathbb{F}_2$-Vektoren und umgekehrt vermieden werden, denn eine Anwendung der beiden „black box"-Operatoren erfordert fast ausschließlich Polynomarithmetik über $\mathbb{F}_q$ und nicht über $\mathbb{F}_2$.

3.2 Das Faktorisierungsmodul

Im Laufe der verschiedenen Faktorisierungsalgorithmen treten sogenannte *Teilfaktorisierungen* oder *partielle Faktorisierungen* des zu zerlegenden Polynoms f auf. Darunter verstehen wir eine Menge M von Paaren (b, i), wobei b ein normierter Teiler von f und $i \in \mathbb{N}$ ist, mit der zusätzlichen Eigenschaft

$$\prod_{(b,i)\in M} b^i = f$$

Wir nennen eine partielle Faktorisierung *teilerfremd*, wenn die Polynome b alle paarweise relativ prim sind. Eine teilerfremde partielle Faktorisierung heißt *vollständig*, wenn alle Faktoren b irreduzibel sind. Im Programm werden Teilfaktorisierungen durch den Datentyp `FactorList` modelliert.

Das eigentliche Faktorisierungsprogramm beinhaltet zwei Teile: einen für die Faktorisierung mittels Gauß-Elimination und einen für Faktorisierung nach Kaltofen und Lobo. Beide Teile bedienen sich gemeinsamer Hilfsfunktionen, z.B. für die quadratfreie Zerlegung. Diese werden zunächst kurz erläutert.

```
void refine(FactorList& factorlist);
void it_sqrt(const Poly2r& f, Poly2r& ret, index_t& power);
void make_sqrfree(FactorList& factorlist);
```

Die Funktion `refine` implementiert die Verfeinerung einer beliebigen Teil-Faktorisierung zu einer teilerfremden Teil-Faktorisierung durch sukzessive ggT-Bildung nach Bach et. al. ([BDS93]).

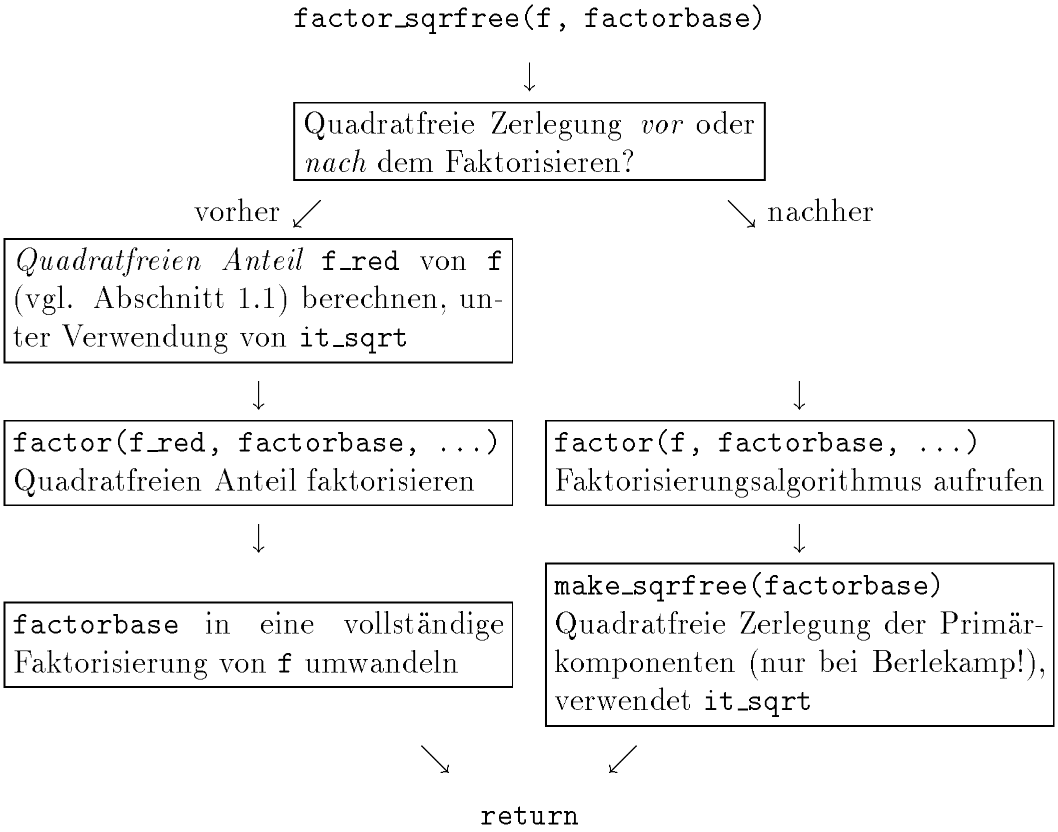
`it_sqrt` („iterated square root") zieht sukzessive Quadratwurzeln aus einem Polynom, so lange das möglich ist. Aus einem Polynom `f` wird ein Polynom `ret` und eine Zweierpotenz `power` mit der Eigenschaft $\texttt{f} = \texttt{ret}^{\texttt{power}}$ und $\texttt{ret}' \neq 0$ berechnet. Dazu muß der Frobenius-Automorphismus $c \mapsto c^2$ von $\mathbb{F}_q = \mathbb{F}_{2^r}$ invertiert werden, was durch $r-1$-malige Anwendung desselben geschieht.

Die Funktion `make_sqrfree` schließlich macht aus einer Teilfaktorisierung in Primärkomponenten (das sind Faktoren mit genau einem irreduziblen Teiler) eine vollständige Faktorisierung, d.h. zerlegt die Primärkomponenten quadratfrei. Dazu verwendet sie zunächst `it_sqrt`, um eine Primärkomponente $g^{2^i u}$ (g irreduzibel, u ungerade) als $(g^u)^{2^i}$ darzustellen, und berechnet anschließend $g := \frac{g^u}{\mathrm{ggT}(g^u,(g^u)')}$ und $u := \deg g^u / \deg g$.

Das Faktorisierungsverfahren wird durch folgende Funktionen implementiert:

```
void factor(const Poly2r& f, FactorList& factorbase, ...);
void factor_sqrfree(const Poly2r& f, FactorList& factorbase);
```

Sie stellen die eigentlichen Faktorisierungsroutinen dar und rufen andere Funktionen als Unterprogramme auf. `factor` faktorisiert das Polynom `f` und liefert die gefundenen Faktoren in `factorbase` zurück. Dabei wird keine quadratfreie Zerlegung vorgenommen, d.h. `factor` liefert im Berlekamp-Fall lediglich die Zerlegung des Polynoms in Primärkomponenten, bei Niederreiter jedoch die vollständige Faktorisierung. Das Verhalten von `factor` hängt davon ab, ob Elemente des Nullraums der jeweiligen Matrix durch Gauß-Elimination oder nach Kaltofen & Lobo mit Wiedemanns Methode berechnet werden. Darauf wird weiter unten genauer eingegangen. `factor_sqrfree` benutzt `factor` und sorgt durch eine quadratfreie Zerlegung vor (oder nach) dem Aufruf von `factor` dafür, daß in jedem Fall die vollständige Faktorisierung berechnet wird. Seine Funktionsweise läßt sich im Flußdiagramm so darstellen:



Die Berechnung des quadratfreien Anteils `f_red` erfolgt so, daß zunächst mit `it_sqrt` eine Darstellung $f = (f_1)^{2^i}$ mit $f_1{}' \neq 0$ und anschließend $f_{\boxtimes} := \frac{f_1}{\mathrm{ggT}(f_1, f_1{}')}$ berechnet wird.

Aus der Faktorisierung von `f_red` wird dadurch die vollständige Faktorisierung von `f` gewonnen, daß mittels sukzessiver Division festgestellt wird, mit welcher Potenz jeder irreduzible Faktor von `f_red` in `f` aufgeht.

Faktorisieren mit Gauß-Elimination

Dazu existieren folgende Funktionen:

```
void setup_B_matrix(const Poly2r& f, Matrix2& B);
void setup_N_matrix(const Poly2r& f, Matrix2& N);
void convert(const Vector2& v, ..., Poly2r& ret);
void factor(const Poly2r& f, FactorList& factorbase, ...);
```

Die Funktionen `setup_B_matrix` und `setup_N_matrix` stellen die zu `f` gehörigen Matrizen $\overline{B}$ bzw. N (bzgl. der Standardbasis (1.9) bzw. (1.21)) aus den Algorithmen von Berlekamp bzw. Niederreiter auf und liefern ihr Ergebnis in `B` bzw.

`N` zurück. Die Funktion `convert` konvertiert einen Vektor `v` über $\mathbb{F}_2$, wie er etwa als Ergebnis der Elimination einer Matrix auftritt, von der Darstellung bzgl. der Standardbasis in ein Polynom `ret` über $\mathbb{F}_q$.

Die Funktion `factor` (s.o.) verhält sich bei der Faktorisierung mit Gauß-Elimination wie folgt:

`factor(f, factorbase, ...)`

↓

`Matrix2 A;`
`setup_B_matrix(f, A)` bzw. `setup_N_matrix(f, A)`
Zu `f` gehörige Matrix aufstellen

↓

`Matrix2* basis = nullspace(A)` Basis des Nullraums von `A` berechnen

↓

`f` durch sukzessive ggT-Bildung mit den Elementen der Basis zerlegen, verwendet `convert` und `refine`

↓

`return`

Die Zerlegung geht so vor sich, daß schrittweise immer feinere partielle Faktorisierungen von `f` berechnet werden. In Pseudocode sieht das so aus:

{ factorbase enthält während des Ablaufs eine partielle Faktorisierung des zu zerlegenden Polynoms f, also eine Menge von Paaren (b, i), wobei b ein Teiler von f und $i \in \mathbb{N}$ ist. Zu Beginn und am Ende jeder Iteration der Schleife II. gilt die Invariante $\prod_{(b,i)\in\text{factorbase}} b^i = f$, und die Faktoren b sind paarweise relativ prim. }

I. Setze factorbase := $\{(f, 1)\}$.

II. Wiederhole die folgenden Schritte für alle Vektoren $\overline{h}$ aus der Nullraumbasis von A:

III. Konvertiere den $\mathbb{F}_2$-Vektor $\overline{h}$ mittels `convert` in ein $\mathbb{F}_q$-Polynom h mit $\deg h < \deg f$, und setze $c := \frac{f}{\mathrm{ggT}(f,h)}$.

IV. Setze oldfactorbase := factorbase und factorbase := $\emptyset$.

V. *(split)* Für alle $(b, i) \in$ oldfactorbase berechne $g := \mathrm{ggT}(b, c)$ und setze factorbase := factorbase $\cup \{(g, i), (\frac{b}{g}, i)\} \setminus \{(1, i)\}$.

VI. Falls f nicht quadratfrei, A die Niederreiter-Matrix und oldfactorbase ungleich factorbase ist, mache mit `refine` die partielle Faktorisierung factorbase zu einer *teilerfremden* partiellen Faktorisierung.

Das Verfahren unterscheidet sich in zwei Punkten von dem der Algorithmen 1.4 und 1.11.

1. Die ggT's mit den Faktoren b der partiellen Faktorisierung werden statt mit h mit dem Cofaktor $c = \frac{f}{\mathrm{ggT}(f,h)}$ gebildet. Sind $h_1, \ldots, h_t$ die Elemente einer Basis von $\overline{\mathcal{B}}$ bzw. $\mathcal{N}$ und $c_1, \ldots, c_t$ die entsprechenden Cofaktoren, dann ist auch $\{c_1, \ldots, c_t\}$ in Charakteristik 2 eine separierende Menge: Im Berlekamp- wie im Niederreiter-Fall gilt für allgemeines (d.h. nicht notwendig quadratfreies) f mit kanonischer Faktorisierung $f = g_1^{e_1} \cdot \ldots \cdot g_1^{e_t}$

   $$\forall (1 \le i < j \le t)\ \exists (1 \le k \le t):\ (g_i^{e_i} | h_k \wedge g_j^{e_j} \nmid h_k) \vee (g_i^{e_i} \nmid h_k \wedge g_j^{e_j} | h_k)$$

   (vgl. Bem. 1.8 und Lemma 1.10). Diese Aussage bleibt gültig, wenn h_k durch $\mathrm{ggT}(f, h_k)$ ersetzt wird. Da c_k der komplementäre Faktor von $\mathrm{ggT}(f, h_k)$ in f ist, gilt

   $$\forall (1 \le i < j \le t)\ \exists (1 \le k \le t):\ (g_i \nmid c_k \wedge g_j | c_k) \vee (g_i | c_k \wedge g_j \nmid c_k)$$

   Im Berlekamp-Fall impliziert das

   $$\forall (1 \le i < j \le t)\ \exists (1 \le k \le t):\ (g_i^{e_i} \nmid c_k \wedge g_j^{e_j} | c_k) \vee (g_i^{e_i} | c_k \wedge g_j^{e_j} \nmid c_k)$$

   d.h. das oben beschriebene Verfahren liefert ebenfalls die Primärkomponenten $g_i^{e_i}$ von f. Im Niederreiter-Fall jedoch sind die c_k quadratfrei, da nach Lemma 1.10 jedes h_k durch $\frac{f}{g_1 \cdot \ldots \cdot g_t}$ teilbar ist. Damit gilt

   $$\forall (1 \le i < j \le t)\ \exists (1 \le k \le t):\ (g_i \nmid c_k \wedge g_j \| c_k) \vee (g_i \| c_k \wedge g_j \nmid c_k) \quad (3.1)$$

   wobei $a \| b$ eine Abkürzung für $a | b \wedge a^2 \nmid b$ ist.

2. Im Niederreiter-Fall für nicht quadratfreies f wird an das „splitting" eine zusätzliche Phase angeschlossen. Sie macht aus der zu diesem Zeitpunkt bestehenden partiellen Faktorisierung eine teilerfremde Teilfaktorisierung, bei der natürlich bereits getrennte irreduzible Faktoren getrennt bleiben (vgl. [BDS93]). Das garantiert für diesen Fall, daß das Verfahren die vollständige Faktorisierung von f liefert:

Da die h_k eine Basis von $\mathcal{N}$ bilden, gibt es zu jedem i mindestens ein k mit $g_i^{e_i} \nmid h_k$ (vgl. Lemma 1.10). Daraus folgt, daß g_i ein Teiler von c_k ist, wegen der Quadratfreiheit von c_k gilt sogar $g_i \| c_k$. Sei b der Faktor der partiellen Faktorisierung factorbase vor dem k-ten Durchlauf von Schritt IV., welcher g_i enthält. Falls noch nicht $g_i \| b$ gilt, wird b in Schritt V. in zwei nicht triviale Faktoren $b_1 = g$ und $b_2 = \frac{b}{g}$ mit $g_i \| b_1$ gespalten. Nach der Verfeinerung der partiellen Faktorisierung in Schritt VI. enthält diese also *genau einen* Faktor $\overline{b}$ mit $g_i \| \overline{b}$, und für alle von $\overline{b}$ verschiedenen Faktoren b gilt $g_i \nmid b$.

Faktorisieren nach Kaltofen & Lobo

Dafür stehen die folgenden Funktionen zur Verfügung:

```
GF2 scalarprod(const Poly2& z1, const Poly2& z2);
GF2 scalarprod(const Poly2r& z1, const Poly2r& z2);
void minpoly(const Poly2r& u, BlackBox<GF2r>& A,
             Poly2& m, index_t& preperiod, index_t maxdegree);
void eval(const Poly2& m, BlackBox<GF2r>& A);
void reon(BlackBox<GF2r>& A, int iteration);
void split(const Poly2r& f, Poly2r& f1, Poly2r& f2);
void refine(FactorList& factorlist);
void factor(const Poly2r& f, FactorList& factorbase, ...);
```

Die beiden `scalarprod`-Funktionen berechnen das Skalarprodukt ihrer Argumente, aufgefaßt als Vektoren über $\mathbb{F}_2$ in der Standard-Polynombasisdarstellung.

Die Funktion `minpoly` implementiert die Berechnung des Minimalpolynoms der Folge $(u^{\mathbf{T}}b, u^{\mathbf{T}}Ab, u^{\mathbf{T}}A^2b, \ldots) \in {\mathbb{F}_2}^\infty$ mit dem Euklidischen Algorithmus (Algorithmus 2.11). Dabei wird in `u` der Vektor u als $\mathbb{F}_q$-Polynom und in `A` eine „black box"-Version der Matrix A über $\mathbb{F}_2$ übergeben, in deren Puffer `A.buffer()` sich der Vektor b als $\mathbb{F}_q$-Polynom befindet. `maxdegree` enthält eine obere Schranke für den Grad des Minimalpolynoms. Das Minimalpolynom der Folge wird in der folgenden Form zurückgeliefert: `m` enthält den X-freien Anteil und `preperiod` die Vielfachheit des Faktors X, d.h. das Minimalpolynom ist $\mathtt{m} \cdot X^{\mathtt{preperiod}}$. Findet `minpoly` bei der Berechnung der Folgenglieder $u^{\mathbf{T}}A^ib$ ein Element des Nullraums von A, so ist dieses nach dem Funktionsaufruf in `A.oldbuffer()` zu finden, und `A.buffer()` enthält das Nullpolynom.

Die Funktion `eval` berechnet $m(A)b$ für ein $\mathbb{F}_2$-Polynom m, eine „black box"-Matrix A über $\mathbb{F}_2$ und einen Vektor (ein Polynom) b. Die Argumente werden in `m` und `A` übergeben, der Vektor b befindet sich als $\mathbb{F}_q$-Polynom im Puffer `A.buffer()` von `A`. Das Ergebnis wird ebenfalls in `A.buffer()` abgelegt. Die Berechnung erfolgt mit einem dem Horner-Schema ähnlichen Verfahren, welches höchstens $\deg m$ Anwendungen von A auf einen Vektor benötigt.

Die Funktion `reon` (Abkürzung für „random element of nullspace") implementiert das Kernstück des Verfahrens von Kaltofen & Lobo, Algorithmus 2.6, mit kleinen Abwandlungen. Es wird ein Element des Nullraums von `A` berechnet und in `A.oldbuffer()` zurückgeliefert. `iteration` gibt an, wie oft `reon` bereits für dieselbe Matrix `A` aufgerufen wurde. Die Funktionsweise in Pseudocode:

`reon(A, iteration)`

{ input: „black box"-Matrix A der Dimension nr über $\mathbb{F}_2$, wobei r der Erweiterungsgrad von $\mathbb{F}_q/\mathbb{F}_2$ ist, iteration $\in \mathbb{N}$ }

{ output: Ein Element des Nullraums von A }

{ Minpoly ist eine statische Variable, die einen Teiler des X-freien Anteils des Minimalpolynoms von A über $\mathbb{F}_2$ enthält. Sie behält ihren Wert zwischen zwei Aufrufen von `reon`, wird beim ersten Aufruf auf 1 gesetzt und bei jedem Folgeaufruf aktualisiert. }

I. Falls iteration $= 0$, setze Minpoly $:= 1$.

II. Wiederhole die folgenden Schritte so lange, bis ein Element des Nullraums von A gefunden ist:

III. Wähle zufällig $0 \neq b \in {\mathbb{F}_2}^{nr}$, und setze $b' :=$ Minpoly$(A)b$ (mittels `eval`).

IV. Wiederhole die Schritte V. und VI., so lange $b' \neq 0$:

V. Wähle zufällig $0 \neq u \in {\mathbb{F}_2}^{nr}$, und berechne mittels `minpoly` das Minimalpolynom der Folge $(u^{\mathbf{T}} A^i b')_{i \geq 0}$, dessen Grad höchstens $nr - \deg$ Minpoly beträgt. Falls dabei ein Element des Nullraums von A gefunden wird, brich die Schleifen IV. und II. ab.

VI. Sei m der X-freie Anteil des in Schritt V. berechneten Minimalpolynoms. Setze Minpoly $:=$ Minpoly $* m$ und (mittels `eval`) $b' := m(A)b'$ $(=$ Minpoly$(A)b)$.

VII. Gib das in Schritt V. gefundene Element des Nullraums aus.

Diese Version unterscheidet sich in folgenden Punkten von Algorithmus 2.6:

1. Es wird nicht notwendig das Minimalpolynom $m_{A,b}$ der Folge $(A^i b)_{i\geq 0}$ berechnet. Statt dessen werden vom Grad her immer größere Teiler des Minimalpolynoms von A berechnet, genauer seines X-freien Anteils, und zwar so lange, bis zur Annullierung von b tatsächlich nur noch eine X-Potenz fehlt. Die Bestimmung eines Nullraums-Elements wird dabei von der Funktion `minpoly` übernommen.

   Ist die Iteration so weit fortgeschritten, daß Minpoly bis auf eine X-Potenz ein Vielfaches von $m_{A,b}$ ist, dann ist das Minimalpolynom der Folge $(A^i b')_{i\geq 0}$ für $b' = \mathrm{Minpoly}(A)b$ genau diese X-Potenz, d.h. es gibt ein kleinstes $d < nr - \deg \mathrm{Minpoly}$ mit $A^d b' = 0$. Im Fall $d > 0$ findet `minpoly` in Schritt V. bei der Berechnung der Folgenglieder $u^{\mathbf{T}} A^i b'$ das Element $0 \neq A^{d-1} b' \in \ker A$. Falls $d = 0$, so ist bereits $b' = 0$, und die Schleife IV. wird abgebrochen, was zur Wahl eines neuen b führt.

2. Der berechnete Teiler Minpoly des Minimalpolynoms von A wird sowohl bei einem „erfolglosen“ b (d.h. das Minimalpolynom von $(A^i b)_{i\geq 0}$ ist nicht durch X teilbar) als auch über zwei Aufrufe von `reon` hinweg aufgehoben. Das hat den Vorteil, daß für jedes neu gewählte b der erste Schritt, nämlich die kostenintensive Berechnung der Folgenglieder $u^{\mathbf{T}} A^i b$ $(0 \leq i < 2rn)$ für die Minimalpolynomberechnung, entfällt. Statt dessen wird gleich mit $b' = \mathrm{Minpoly}(A)b$ weiter iteriert, und je größer der Grad von Minpoly bereits ist, um so wahrscheinlicher ist das Minimalpolynom von $(A^i b')_{i\geq 0}$ nur noch eine X-Potenz.

Die Funktion `split` schließlich versucht, das Polynom `f` in zwei nicht triviale Faktoren `f1` und `f2` zu zerlegen. Dazu wird, je nach Einstellung der Parameter für die Faktorisierung, mittels `reon` ein Polynom $\neq 0$ im Nullraum des Berlekamp-Operators $\overline{\beta}$ bzw. des Niederreiter-Operators ν und anschließend dessen größter gemeinsamer Teiler mit `f` berechnet. Falls der ggT trivial ist, wird das so oft wiederholt, bis der ggT nicht trivial oder eine vorgegebene Anzahl von Versuchen überschritten ist (die Schranke m in Algorithmus 2.7). Im letzten Fall wird das Polynom `f` als irreduzibel angenommen, was es tatsächlich nur mit einer gewissen Wahrscheinlichkeit ist (vgl. Abschnitt 2.3). Übrigens führt im nicht quadratfreien Niederreiter-Fall *jedes* von 0 verschiedene Element des Nullraums von ν zu einem nicht trivialen ggT (vgl. Lemma 1.10 und Bem. 1.17).

`refine` wurde schon weiter oben beschrieben, und `factor` verhält sich bei der Variante des Faktorisierungsverfahrens nach Kaltofen & Lobo wie folgt:

`factor(f, factorbase, ...)`

↓

`Poly2r f1, f2;`
`split(f, f1, f2);`
`f` in zwei Faktoren zerlegen. Falls trivialer „split“: `f` als irreduzibel deklarieren und STOP.

↓

Generiere partielle Faktorisierung `factorbase`, bestehend aus den beiden Faktoren `f1`, `f2`, jeweils in der ersten Potenz. Falls `f` nicht quadratfrei ist und nach Niederreiter faktorisiert wird, die partielle Faktorisierung `factorbase` mit `refine` verfeinern.

↓

Rekursive Anwendung des Verfahrens auf alle Faktoren aus `factorbase`.

↓

`return`

3.3 Bedienungsanleitung

Das Programm `factor2` ist die Benutzeroberfläche für das Faktorisierungsprogramm. Es gestattet die Einstellung verschiedener Parameter für Arithmetik und Faktorisierung, die Auswahl bzw. Eingabe eines zu faktorisierenden Polynoms, die Protokollierung der Berechnung und die Messung der verbrauchten Systemressourcen. Das Programmpaket umfaßt die Dateien

`factor2`	ausführbares Programm
`factor2.help`	Online-Hilfetext
`irreducible.list`	Datei mit irreduziblen Polynomen über $\mathbb{F}_2$ bis Grad 200

Nach dem Aufruf mit

```
factor2
```

wird zunächst gefragt, ob ein Hilfetext angezeigt werden soll. Falls nicht, so ist der Erweiterungsgrad r von $\mathbb{F}_q/\mathbb{F}_2$ einzugeben und anschließend ein generierendes Polynom vom Grad r für diese Körpererweiterung (siehe unter „Ein- und Ausgabe von Polynomen“). Durch einfaches Drücken der RETURN-Taste kann auch, falls möglich, das generierende Polynom aus der Datei `irreducible.list` eingelesen werden. Anschließend wird das folgende Menü angezeigt:

```
Faktorisieren von Polynomen ueber GF(2^r)
1) Zu faktorisierendes Polynom eingeben
2) Polynom(e) mit zufaellig gewaehlten Koeffizienten faktorisieren
6) Hilfetext
7) Parameter fuer die Polynom-Arithmetik einstellen
8) Parameter fuer das Faktorisierungsverfahren einstellen
9) Anderen Koerper waehlen
0) Programm beenden
?
```

Unter 1) kann ein Polynom über $\mathbb{F}_q$ eingegeben werden, welches anschließend faktorisiert wird. Zusätzlich kann angegeben werden, daß das Verfahren mehrfach hintereinander auf dasselbe Polynom angewandt werden soll (etwa um durch probabilistische Algorithmen bedingte Laufzeitschwankungen zu messen). Danach kann ausgewählt werden, ob die Berechnung auf dem Bildschirm protokolliert und Ergebnisse und/oder eine Statistik der benötigten Körperoperationen in $\mathbb{F}_q$ sowie der verbrauchten System-Ressourcen CPU-Zeit und Speicherplatz ausgegeben werden soll.

Bei 2) können Zufallspolynome über $\mathbb{F}_q$ mit beschränktem Grad faktorisiert werden. Hier ist zunächst der Grad anzugeben, anschließend die Anzahl *verschiedener*, unabhängig gewählter Zufallspolynome, die nacheinander faktorisiert werden sollen. Danach ist eine Auswahl wie unter 1) zu treffen.

Menüpunkt 6) dient zur Ausgabe eines Online-Hilfetextes.

Unter 7) werden verschiedene Parameter für die Polynomarithmetik sowohl in $\mathbb{F}_2[X]$ (welche der Arithmetik in $\mathbb{F}_q$ zugrunde liegt) als auch in $\mathbb{F}_q[X]$ eingestellt: Die Art der zu verwendenden Algorithmen für Multiplikation (klassisch oder Karatsuba), Polynomdivision und ggT-Berechnung, sowie Gradschranken für deren Anwendung.

Punkt 8) gestattet die Einstellung verschiedener Parameter für das Faktorisierungsverfahren selbst: Berlekamp oder Niederreiter, Gauß-Elimination oder Kaltofen & Lobo einschließlich Gradschranken für deren Anwendung, quadratfreie Zerlegung vor oder nach dem Faktorisieren und die maximale Anzahl von Versuchen bei Kaltofen & Lobo, um ein Polynom zu „splitten“.

Mit 9) ist es möglich, einen anderen Körper $\mathbb{F}_q$ zu wählen.

Ein- und Ausgabe von Polynomen

Auf dem Bildschirm werden $\mathbb{F}_2$- und $\mathbb{F}_q$-Polynome in zwei Arten dargestellt. Bei der Ausgabe heißt die Unbestimmte für $\mathbb{F}_2$-Polynome `x`, für $\mathbb{F}_q$-Polynome hingegen `y`. Die Koeffizienten von $\mathbb{F}_q$-Polynomen werden ihrerseits wieder als polynomiale Ausdrücke kleinsten Grades in `x` dargestellt, wobei `x` ein primitives Element der Körpererweiterung $\mathbb{F}_q/\mathbb{F}_2$ repräsentiert, also eine Nullstelle des generierenden Polynoms. Polynome werden stets nach aufsteigendem Grad sortiert ausgegeben. Die Darstellung ist mit der von MAPLE für Polynome kompatibel.

Beispiel: Das Polynom $1 + X + X^3 + X^7 + X^{10} \in \mathbb{F}_2[X]$ wird als

```
1+x+x^3+x^7+x^10
```

ausgegeben. Das Polynom

$$\alpha + (1 + \alpha^2)X + (\alpha + \alpha^2)X^3 + X^4 \in \mathbb{F}_q[X] \tag{3.2}$$

für $q = 2^4$ und $\mathbb{F}_q = \mathbb{F}_2(\alpha)$ wird als

```
(x)+(1+x^2)*y+(x+x^2)*y^3+y^4
```

ausgegeben.

Für die Eingabe von $\mathbb{F}_2$- und $\mathbb{F}_q$-Polynomen wurde eine kürzere Darstellung gewählt, die auf die explizite Angabe von `x` und `y` verzichtet. Ein $\mathbb{F}_2$-Polynom (oder ein polynomialer Ausdruck über $\mathbb{F}_2$, z.B. ein Element von $\mathbb{F}_q$) in x wird als Folge seiner Indizes i angegeben, für die der Koeffizient von x^i ungleich 0 ist.

Beispiel: Das Polynom $1 + x + x^3 + x^7 + x^{10}$ wird eingegeben als

```
0 1 3 7 10
```

Die Reihenfolge spielt dabei keine Rolle, d.h.

```
10 1 3 7 0
```

stellt dasselbe Polynom dar.

Für die Eingabe von $\mathbb{F}_q$-Polynomen gilt sinngemäß dasselbe wie bei $\mathbb{F}_2$-Polynomen, nur daß zusätzlich der jeweilige Koeffizient vor dem Index in Klammern angegeben werden muß.

Beispiel: Das Polynom (3.2) wird eingegeben als

```
(1)0 (0 2)1 (1 2)3 (0)4
```

Auch hier spielt die Reihenfolge keine Rolle. Koeffizienten $= 1$ können weggelassen werden, d.h. obiges Polynom kann auch als

```
(1)0 (0 2)1 (1 2)3 4
```

eingegeben werden.

Einstellung der Parameter für Polynomarithmetik und Faktorisierung

Wird im Menü der Punkt 7) aufgerufen, dann sind für die Arithmetik in $\mathbb{F}_2[X]$ und in $\mathbb{F}_q[X]$ nacheinander und unabhängig voneinander folgende Einstellungen möglich: Zunächst ist auszuwählen, welches Verfahren für die Multiplikation verwendet werden soll, die klassische quadratische Methode oder das asymptotische schnellere „divide and conquer"-Verfahren nach Karatsuba. Nur wenn die Karatsuba-Multiplikation gewählt wird, sind noch je eine Gradschranke für Multiplikation, Division und ggT-Berechnung einzugeben. Für Polynome, deren Grad oberhalb der jeweiligen Gradschranke liegt, werden die asymptotisch schnelleren Algorithmen Karatsuba-Multiplikation, Polynomdivision durch Newton-Inversion bzw. schnelle ggT-Berechnung (vgl. Abschnitt 3.1) verwendet. Für Polynome unterhalb dieser Gradschranke, wie sie z.B. auch im Verlauf der Rekursion der schnellen Algorithmen auftreten, wird auf die klassischen Verfahren umgeschaltet. Die vom Programm vorgeschlagenen Default-Werte für diese Gradschranken sind annähernd optimal.

Unter dem Menüpunkt 8) können die Parameter für das Faktorisierungsverfahren eingestellt werden. Zunächst ist die Frage zu beantworten, ob nach Berlekamp oder nach Niederreiter zu faktorisieren ist. Anschließend kann eingestellt werden, ob vor oder nach dem Aufruf des Faktorisierungsalgorithmus quadratfrei zerlegt werden soll (bei Niederreiter ist eine quadratfreie Zerlegung hinterher überflüssig). Schließlich fragt das Programm nach der Art des zu verwendenden Algorithmus für die Bestimmung von Elementen aus dem Nullraum der jeweiligen Matrix. Hier kann zwischen der deterministischen Gauß-Elimination der Algorithmen 1.4 und 1.11 und der probabilistischen Methode nach Wiedemann, wie sie von Kaltofen & Lobo in Algorithmus 2.7 verwendet wird, gewählt werden. Für die letzte Variante sind noch zwei weitere Parameter wichtig: eine Gradschranke für die Umschaltung auf Gauß-Elimination während des rekursiven Faktorisierungsprozesses, und die maximale Anzahl von Versuchen, ein Polynom zu „splitten", bevor es als irreduzibel angenommen wird (m in Algorithmus 2.7). Je größer diese Anzahl gewählt wird, um so länger ist die Laufzeit des Verfahrens und um so größer die Wahrscheinlichkeit, mit der der Algorithmus die vollständige Faktorisierung berechnet (vgl. die probabilistische Analyse in Abschnitt 2.3).

3.4 Testläufe und Bewertung

In den Testläufen wurden Zufallspolynome für verschiedene Erweiterungsgrade r von $\mathbb{F}_q$ über $\mathbb{F}_2$ und Polynomgrade n faktorisiert. Die Tabellen geben die dafür benötigten Ressourcen CPU-Zeit und Speicherplatz an. Bei allen angegebenen

Testläufen wurde vor dem Faktorisieren eine quadratfreie Zerlegung des Polynoms vorgenommen. Die Messungen wurden auf einer SUN Sparc Station 10/20 mit einer Taktrate von 33 MHz und 32 MB Hauptspeicher vorgenommen.

In der ersten Tabelle sind die Daten für festen Erweiterungsgrad $r = 10$ und variablen Polynomgrad n dargestellt. Sie wurden gemessen mit den Algorithmen 1.4 (Berlekamp) in der absoluten Version und 1.11 (Niederreiter), beide mit Gauß-Elimination, und mit Maple's `Factor mod 2`, welches den relativen Berlekamp-Algorithmus verwendet.

n	Berlekamp		Niederreiter		Maple V.2	
10	2 sec.	572 K	1.5 sec.	580 K	3 sec.	917 K
20	15 sec.	664 K	10 sec.	672 K	18 sec.	1441 K
50	246 sec.	1304 K	158 sec.	1312 K	302 sec.	1703 K
100	1928 sec.	3536 K	1226 sec.	3420 K	2506 sec.	2096 K
200	15985 sec.	12448 K	10222 sec.	12232 K	29815 sec.	3407 K

Der weitaus größte Anteil der Kosten entfällt bei den Algorithmen 1.4 und 1.11 auf die Elimination der $nr \times nr$-Matrix. Die Laufzeitdifferenzen ergeben sich vermutlich daraus, daß die Matrix N in Charakteristik 2 relativ „dünn" ist, was für die Matrix $\overline{B}$ nicht unbedingt der Fall ist, und das verwendete Eliminationsverfahren deshalb im Niederreiter-Fall schneller läuft als im Berlekamp-Fall.

Die zweite Tabelle gibt die Daten der Testläufe für festen Polynomgrad $n = 10$ und variablen Erweiterungsgrad r an, gemessen mit denselben Algorithmen wie oben.

r	Berlekamp		Niederreiter		Maple V.2	
10	2 sec.	572 K	1.5 sec.	580 K	3 sec.	917 K
20	11 sec.	652 K	9 sec.	660 K	5 sec.	1245 K
50	167 sec.	1184 K	136 sec.	1192 K	28 sec.	1376 K
100	1320 sec.	3016 K	1105 sec.	2948 K	138 sec.	1572 K
200	10702 sec.	10208 K	8850 sec.	10132 K	158 sec.	1834 K

Sinngemäß gilt für diese Tabelle dasselbe wie oben. Der relative Berlekamp-Algorithmus, den Maple verwendet, erweist sich für kleine Polynomgrade als deutlich schneller, da nur eine $n \times n$-Matrix über $\mathbb{F}_q$ statt einer $nr \times nr$-Matrix über $\mathbb{F}_2$ eliminiert werden muß.

Die nächste Tabelle vergleicht die Laufzeiten und den Speicherbedarf der klassischen deterministischen Algorithmen, die die Gauß-Elimination verwenden, mit denen der Algorithmen a la Kaltofen & Lobo, die auf Wiedemanns probabilistischer Methode beruhen, für festen Erweiterungsgrad $r = 10$ und festen Polynomgrad $n = 200$. BG bzw. NG bezeichnen dieselben Algorithmen wie oben, d.h. den Berlekamp-Algorithmus 1.4 in der absoluten Version bzw. den Niederreiter-Algorithmus 1.11 (das „G“ steht für „Gauß“). BW bzw. NW bezeichnen den Algorithmus 2.7 nach Kaltofen & Lobo bzw. die Niederreiter-Variante desselben („W“ steht für „Wiedemann“).

Algorithmus	Laufzeit	Speicherplatz
BG	15985 sec.	12448 K
NG	10222 sec.	12232 K
BW	286526 sec.	1560 K
NW	118667 sec.	732 K
Maple	29815 sec.	3407 K

Die beiden Algorithmen BW und NW verwenden die Karatsuba-Multiplikation für $\mathbb{F}_q$-Polynome vom Grad > 10, die schnelleren Polynomdivisions- und ggT-Algorithmen rentieren sich bei dieser Größenordnung noch nicht. Wird der Grad der im Verlauf des rekusiven Verfahrens auftretenden Polynome kleiner als 100, dann schalten die Algorithmen um auf Gauß-Elimination. Es werden bis zu 5 Versuche gemacht, ein Polynom in zwei nichttriviale Faktoren zu zerlegen, bis das Polynom als irreduzibel angenommen wird.

Wie in Abschnitt 2.3 beschrieben, ist das Ergebnis der beiden Algorithmen BW und NW nur mit einer gewissen Wahrscheinlichkeit die vollständige Faktorisierung des Polynoms. Mit $m = 5$ und der Gradschranke von 100 kann im Testlauf BW für diese Wahrscheinlichkeit eine untere Schranke von $1-2^{-5}$ angegeben werden, da höchstens ein Faktor vom Grad ≥ 100 fälschlicherweise als irreduzibel angenommen werden kann; für Faktoren vom Grad < 100 liefert der deterministische Berlekamp-Algorithmus in jedem Fall das korrekte Ergebnis. Für *NW* ist nach den Analysen in Abschnitt 2.3 keine so gute Schranke möglich.

Die Daten zeigen, daß die probabilistischen Algorithmen BW und NW in dieser Implementation für die angegebenen Werte von r und n deutlich langsamer sind als die deterministischen Algorithmen BG und NG. Der größte Anteil der Laufzeit der Algorithmen BW und NW fällt für die „black box“-Anwendungen der jeweiligen Matrizen $\overline{B}$ bzw. N ab. Eine Anwendung von $\overline{B}$ bzw. N auf einen Vektor für $r = 10$ und $n = 200$ mit den oben angegebenen Einstellungen für die Polynomarithmetik dauert 27.53 sec. bzw. 4.77 sec., und für den ersten „split“ sind alleine schon rund 4000 „black box“-Anwendungen erforderlich. Die Niederreiter-Variante mit der Beschleunigung nach [GvzG94] (vgl. Abschnitt 3.1) für eine „black box“-Anwendung erweist sich als schneller, dafür ist die Faktorisierung mit einer möglicherweise sehr großen Wahrscheinlichkeit nicht vollständig.

Die in dieser Arbeit vorgenommene Anpassung des Algorithmus von Kaltofen & Lobo für die Niederreiter-Matrix scheint also für die Praxis wegen der unwägbaren Fehlerwahrscheinlichkeit untauglich. Die Messungen zeigen aber, daß die Verwendung der Niederreiter-Matrix in Faktorisierungsalgorithmen, die auf der Methoden von Kaltofen & Saunders ([KS91]) beruhen (siehe [GvzG94]), durchaus Laufzeit-Vorteile gegenüber der Berlekamp-Variante hat.

Literaturverzeichnis

[AHU75] A. V. Aho, J. E. Hopcroft und J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley 1975.

[BDS93] E. Bach, J. Driscoll und J. Shallit, Factor refinement, *Journal of algorithms* **15**(1993) pp. 199–222.

[Ber68] E. R. Berlekamp, *Algebraic Coding Theory*, McGraw-Hill New York 1968.

[BGY80] R. P. Brent, F. G. Gustavson und D. Y. Y. Yun, Fast solution of Toeplitz systems of equations and computation of Padé approximants, *Journal of Algorithms* **1**(1980) pp. 259–292.

[Bla85] R. E. Blahut, *Fast algorithms for digital signal processing*, Addison-Wesley 1985.

[Bla92] R. E. Blahut, *Algebraic methods for signal processing and communications*, Springer 1992.

[Dor87] J. L. Dornstetter, On the equivalence between Berlekamp's and Euclid's algorithms, *IEEE Transactions on Information Theory* **IT-33**(1987) pp. 428–431.

[ES90] M. A. Ellis und B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley 1990.

[Fle93] P. Fleischmann, Connections between the algorithms of Berlekamp and Niederreiter for factoring polynomials over $\mathbb{F}_q$, *Lin. Alg. Appl.* **192**(1993) pp. 101–108.

[GCL92] K. O. Geddes, S. R. Czapor und G. Labahn, Polynomial factorization, Kap. 8 in: *Algorithms for computer algebra*, Kluwer 1992.

[GvzG94] S. Gao und J. von zur Gathen, Berlekamp's and Niederreiter's polynomial factorization algorithms, in: G. L. Mullen und P. J.-S. Shiue, Hrsg., *Finite Fields: Theory, Applications and Algorithms* (Contemporary Mathematics), Amer. Math. Soc. 1994, Vorabdruck.

[IR82] K. Ireland und M. Rosen, *A Classical Introduction to Modern Number Theory*, Springer-Verlag Berlin 1982.

[Kal90] E. Kaltofen, Polynomial factorization 1982–1986, in: D. V. Chudnovsky und R. D. Jenks, Hrsg., *Computers in Mathematics* (Lecture Notes in Pure and Applied Mathematics 125) pp. 285–309, Marcel Dekker, Inc. New York 1990.

[Kal92] E. Kaltofen, Polynomial factorization 1987–1991, in: *Proc. Latin '92* (LNCS 583) pp. 294–313, 1992.

[KL94] E. Kaltofen und A. Lobo, Factoring high-degree polynomials by the black box Berlekamp algorithm, Technical report, Department of Computer Science, Rensselaer Polytechnic Institute 1994.

[KS91] E. Kaltofen und B. D. Saunders, On Wiedemann's method of solving sparse linear systems, in: *Proc. AAECC-10* (LNCS 539) pp. 29–38, Springer 1991.

[Kun74] H. T. Kung, On computing reciprocals of power series, *Numer. Math.* **22**(1974) pp. 341–348.

[LN83] R. Lidl und H. Niederreiter, *Finite Fields*, Bd. 20 von *Encyclopedia Of Mathematics and Its Applications*, Addison Wesley 1983.

[Mas69] J. L. Massey, Shift register synthesis and BCH decoding, *IEEE Transactions on Information Theory* **IT-15**(1969) pp. 122–127.

[MBG+93] A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullin, S. A. Vanstone und T. Yaghoobian, *Applications of Finite Fields*, Kluwer 1993.

[Mig92] M. Mignotte, *Mathematics for Computer Algebra*, Springer 1992.

[Mil92] V. S. Miller, On the factorization method of Niederreiter, Vorabdruck 1992.

[Moe73] R. T. Moenck, Fast computation of GCDs, in: *Proceedings Fifth Annual Symposium on Theory of Computing* pp. 142–151, 1973.

[MV90] K. Meyberg und P. Vachenauer, *Höhere Mathematik*, Bd. 1, Springer 1990.

[Nie93a] H. Niederreiter, Factorization of polynomials and some linear-algebra problems over finite fields, *Lin. Alg. Appl.* **192**(1993) pp. 301–328.

[Nie93b] H. Niederreiter, New deterministic factorization algorithms for polynomials over finite fields, Vorabdruck 1993.

[Nie93c] H. Niederreiter, A new efficient factorization algorithm for polynomials over small finite fields, *AAECC* **4**(1993) pp. 81–87.

[Nie94] H. Niederreiter, Factoring polynomials over finite fields using differential equations and normal bases, *Mathematics of Computation* **62**(1994) pp. 819–830.

[Rab80] M. O. Rabin, Probabilistic algorithms in finite fields, *SIAM Journal of Computing* **9**(1980) pp. 273–280.

[SKHN75] Y. Sugiyama, M. Kasahara, S. Hirasawa und T. Namekawa, A method for solving key equation for decoding Goppa codes, *Information and Control* **27**(1975) pp. 87–99.

[vzGS92] J. von zur Gathen und V. Shoup, Computing Frobenius maps and factoring polynomials, *Computational Complexity* **2**(1992) pp. 187–224.

[Wie86] D. H. Wiedemann, Solving sparse linear equations over finite fields, *IEEE Transactions on Information Theory* **IT-32**(1986) pp. 54–62.

[WS79] L. R. Welch und R. A. Scholtz, Continued fractions and Berlekamp's algorithm, *IEEE Transactions on Information Theory* **IT-25**(1979) pp. 19–27.